Swinging Types At Work

Peter Padawitz
padawitz@cs.uni-dortmund.de
http://ls5.cs.uni-dortmund.de/∼peter
University of Dortmund

July 25, 2022

Abstract

We present a number of swinging specifications with visible and/or hidden components, such as lists, sets, bags, maps, monads, streams, trees, graphs, processes, nets, classes, languages, parsers,... They provide more or less worked-out case studies and shall allow the reader to figure out the integrative power of the swinging type approach with respect to various specification and proof formalisms. For instance, the translation of algebraic nets into swinging types admits the generalization of net proof methods and thus—via a compiling graph grammar—for verifying SDL specifications. Similarly, UML class diagrams and state machines are turned into swinging types in order to make them amenable to constraint solving and proving.

Contents

All specifications presented in the paper follow the syntax of swinging types. The main actual definitions can be found in [78]: Def. 1.1 (swinging signature), Def. 1.2 (basic Horn specification, swinging type), Def. 1.5 (Herbrand model, functional, continuous, behaviorally consistent specification), Def. 3.1.1 (cosignature), Def. 3.1.7 (cospecification), Def. 3.2.1 (coalgebraic swinging types), Def. 5.1 (coinductive specification). Roughly said, a coalgebraic swinging type is a swinging type built up from a cospecification. Coinductive, functional and continuous specifications are behaviorally consistent ([78], Thm. 5.4; previous version: [75], Thm. 6.5). Criteria for functionality and continuity are given in [74], [76], Chapter 5, and [75], Section 5.

1 Standard types

1.1 Numbers and Booleans

Natural number arithmetic is presented in terms of a basic Horn specification:

NAT

`sorts`	nat
`constructs`	$0 :\to nat$
	$\_ + 1 : nat \to nat$
`defuncts`	$\_ + \_ : nat \times nat \to nat$
	$\_ - \_ : nat \times nat \to nat$
	$min : nat \times nat \to nat$
`static preds`	$\_ \not\equiv \_ : nat \times nat$
	$\_ < \_ : nat \times nat$
	$\_ > \_ : nat \times nat$
	$\_ \leq \_ : nat \times nat$
`vars`	$x, y : nat$
`Horn axioms`	$0 + x \equiv x$
	$(x+1) + y \equiv (x+y) + 1$
	$0 - x \equiv 0$
	$(x+1) - y \equiv (x-y) + 1$
	$min(x,x) \equiv x$
	$min(x,y) \equiv x \quad \Leftarrow \quad x < y$
	$min(x,y) \equiv y \quad \Leftarrow \quad x > y$
	$0 \not\equiv x+1$
	$x+1 \not\equiv 0$
	$x+1 \not\equiv y+1 \quad \Leftarrow \quad x \not\equiv y$
	$0 < x+1$
	$x+1 < y+1 \quad \Leftarrow \quad x < y$
	$x > y \quad \Leftarrow \quad y < x$
	$x \leq x$
	$x \leq y \quad \Leftarrow \quad x < y$

A swinging type with empty visible subspecification provides another presentation of natural number arithmetic:

HNAT

`hidsorts`	nat
`constructs`	$0, 1 :\to nat$
	$\_ + \_ : nat \times nat \to nat$

	$\_-\_ : nat \times nat \to nat$
`destructs`	$pred : nat \to 1 + nat$
`vars`	$x, y, z : nat$
`Horn axioms`	$pred(0) \equiv ()$
	$pred(1) \equiv (0)$
	$pred(x+y) \equiv () \;\Leftarrow\; pred(x) \equiv () \wedge pred(y) \equiv ()$
	$pred(x+y) \equiv (z) \;\Leftarrow\; pred(x) \equiv () \wedge pred(y) \equiv (z)$
	$pred(x+y) \equiv (z+y) \;\Leftarrow\; pred(x) \equiv (z)$
	$pred(x-y) \equiv () \;\Leftarrow\; pred(x) \equiv ()$
	$pred(x-y) \equiv (z-y) \;\Leftarrow\; pred(x) \equiv (z)$

Note that these axioms are coinductive if one uses the extended definition of observing atoms given in Section 1 (cf. [75], Def. 6.1).

NAT and HNAT are dual to each other: the inverse of NAT's *nat*-constructor *succ* provides HNAT's *nat*-destructor *pred*. The initial NAT-model is isomorphic to the final HNAT-model. Both are isomorphic to $\mathbb{N}$. In contrast to NAT, HNAT can be extended easily to a specification of $\mathbb{N} \uplus \{\infty\}$:

NAT_∞ = HNAT **then**

`constructs`	$\infty : \to nat$
`Horn axioms`	$pred(\infty) \equiv (\infty)$

The final NAT_∞-model is isomorphic to the final F-coalgebra $\overline{\mathbb{N}}$ where $F(A) =_{def} 1 + A$. More precisely, $\overline{\mathbb{N}}$ and $1 + \overline{\mathbb{N}}$ are isomorphic to the *nat*- resp. nat'-carrier of $Fin(\mathrm{NAT}_\infty)$. Dually, $\mathbb{N}$ is the initial F-algebra and isomorphic to the initial NAT-model (see Section 6).

Let us show that $x + \infty \sim \infty$ is an inductive theorem of NAT_∞. For the rules applied here, see [76, 78].

$\forall\, x : x + \infty \sim \infty$

coinduction on $\sim_{nat}$

$\vdash\; \exists\, q\; \forall\, x : q(x+\infty, \infty) \;\wedge\; \forall y, z : (q(y,z) \Rightarrow q'(pred(y), pred(z)))$

define q by the axioms $q(x+\infty,\infty)$, $q(\infty,\infty)$ and unfold q

$\vdash\; \forall\, x, y, z : ((y \equiv x + \infty \wedge z \equiv \infty) \Rightarrow q'(pred(y), pred(z))) \;\wedge$
$\quad \forall y, z : ((y \equiv \infty \wedge z \equiv \infty) \Rightarrow q'(pred(y), pred(z)))$

variable elimination

$\vdash\; \forall\, x : q'(pred(x+\infty), pred(\infty)) \;\wedge\; q'(pred(\infty), pred(\infty))$

unfold *pred*

$\vdash\; \forall\, x : q'(pred(x+\infty), (\infty)) \;\wedge\; q'((\infty), (\infty))$

unfold q' with the implicit axioms of q' (cf. [78], Section 1)

$\vdash\; \forall\, x : q'(pred(x+\infty), (\infty)) \;\wedge\; q(\infty, \infty)$

unfold q

$\vdash\; \forall\, x : q'(pred(x+\infty), (\infty))$

unfold *pred*

$\vdash\; \forall\, x : (\exists\, z : q'((z), (\infty)) \wedge pred(x) \equiv () \wedge pred(\infty) \equiv (z)) \;\vee\; (\exists\, y : q'((y+\infty), (\infty)) \wedge pred(x) \equiv (y))$

unfold *pred*

$\vdash\; \forall\, x :\; (\exists\, z : q'((z), (\infty)) \wedge pred(x) \equiv () \wedge (\infty) \equiv (z)) \;\vee\; (\exists\, y : q'((y+\infty), (\infty)) \wedge pred(x) \equiv (y))$

constructor elimination

$\vdash\; \forall\, x :\; (\exists\, z : q'((z), (\infty)) \wedge pred(x) \equiv () \wedge \infty \equiv z) \;\vee\; (\exists\, y : q'((y+\infty), (\infty)) \wedge pred(x) \equiv (y))$

variable elimination

$\vdash\; \forall\, x : (q'((\infty), (\infty)) \wedge pred(x) \equiv ()) \;\vee\; (\exists\, y : q'((y+\infty), (\infty)) \wedge pred(x) \equiv (y))$

unfold q'

$\vdash \ \forall\, x : \ (q(\infty,\infty) \wedge pred(x) \equiv ()) \ \vee \ (\exists\, y : q(y+\infty,\infty) \wedge pred(x) \equiv (y))$

unfold q

$\vdash \ \forall\, x : \ (pred(x) \equiv () \ \vee \ \exists\, y : pred(x) \equiv (y))$

expansion with $pred(x) \equiv () \vee \exists\, y : pred(x) \equiv (y)$

$\vdash \ True$

Boolean arithmetic is also presented in terms of a basic Horn specification:

BOOL

`sorts`	$bool$	
`constructs`	$true, false :\to bool$	
`defuncts`	$not : bool \to bool$	
	$and, or, eq : bool \times bool \to bool$	
`static preds`	$\_ \not\equiv \_ : bool \times bool$	
`vars`	$b, c : bool$	
`Horn axioms`	$not(true) \equiv false$	$true\ and\ b \equiv b$
	$not(false) \equiv true$	$false\ and\ b \equiv false$
	$true\ or\ b \equiv true$	$true \not\equiv false$
	$false\ or\ b \equiv b$	$false \not\equiv true$
	$eq(true, b) \equiv b$	
	$eq(false, b) \equiv not(b)$	

The following specification of integer numbers represents the numbers as terms constructed from 0, 1, + and -. Since the induced structural equivalence is too fine for representing the equality of integers, we specify this equality as a behavioral one with three destructors: successor, predecessor and a test on zero.

INT = BOOL `and`

`hidsorts`	int
`constructs`	$0, 1 :\to int$
	$\_ + \_ : int \times int \to int$
	$\_ - \_ : int \times int \to int$
`destructs`	$pred, succ : int \to int$
	$zero : int$
`vars`	$x, y : int$
`Horn axioms`	$succ(0) \equiv 1$
	$succ(1) \equiv 1 + 1$
	$succ(x + y) \equiv (x + y) + 1$
	$succ(x - y) \equiv (x - y) + 1$
	$pred(0) \equiv 0 - 1$
	$pred(1) \equiv 0$
	$pred(x + y) \equiv (x + y) - 1$
	$pred(x + y) \equiv (x - y) - 1$
	$zero(0)$

The destructor *zero* cannot be dropped. Otherwise all *int*-terms were behaviorally equivalent. The domain completion of INT contains additional axioms $succ(i) \equiv i + 1$ and $pred(i) \equiv i - 1$ for each $i \in \mathbb{Z}$.

Exercise. Show that $(x + y) - y \sim x$ and $(x - y) + y \sim x$ are inductive theorems of INT!

1.2 Generic types

We specify frequently used generic data types, namely lists, sets, bags (or multisets) and maps (or arrays). We use parameter specifications similarly to Haskell type classes each of which is associated with some sort variable s [45]. A type class $TC(s)$ consists of functions and predicates that are polymorphic in s and axioms that restrict the instances of s to those sorts for which corresponding instances of the functions and predicates exist and meet the axioms. $TC(s)$ in the list of base specifications of another specification SP stands for all those instances used in SP. In this way, SP becomes *generic* type. For example, the parameter ENTRY(s) (see below) demands an inequality for s and the subsequent generic type LIST has ENTRY(*entry*) in the list of base specifications and thus may use all sorts, functions and predicates that are polymorphic in *entry* (like *list*(*entry*)) or in instances of *entry* (like *list*(*list*(*entry*))) provided that inequalities for the latter can be derived from the presupposed inequality for *entry*.

This concept of *parametric polymorphism* makes signature morphisms superfluous as a means for instantiating parameter specifications, but of course not as a means for structuring specifications vertically by refinement or data type change.

We use some CASL notations [18]: "`and`" builds the non-disjoint union of specifications and thus identifies synonymous, equally-typed symbols of the argument specifications. "`then`" denotes the *extension* operator that combines a specification with additional signature symbols and axioms.

ENTRY(s) = BOOL `then`

`functs`	$eq : s \times s \to bool$
`preds`	$\_ \not\equiv \_ : s \times s$
`vars`	$x, y : s$
`axioms`	$x \not\equiv y \Leftrightarrow \neg(x \equiv y)$
	$eq(x,x) \equiv true$
	$eq(x,y) \equiv false \quad \Leftarrow \quad x \not\equiv y$

1.2.1 Lists and binary trees

LIST = ENTRY(*entry*) `and` ENTRY(*entry*′) `and` NAT `then`

`hidsorts`	$list = list(entry) \quad list' = list(entry')$
`objconstructs`	$[] : \to list$
	$\_ : \_ : entry \times list \to list$
`defuncts`	$eq : entry \times entry \to bool$
	$[\_] : entry \to list$
	$\_ ++ \_ : list \times list \to list$
	$\_`join`\_ : list \times list \to list$
	$\_ - \_ : list \times list \to list$
	$\_!!\_ : list \times nat \to 1 + entry$
	$drop : nat \times list \to list$
	$sublist : list \to (nat \times nat \to list)$
	$flatten : list(list) \to list$
	$mklist : entry^* \to list$
	$map : (entry \to entry') \to (list \to list')$
	$filter : (entry \to bool) \times list \to list$
	$length : list \to nat$
	$card : list \times entry \to nat$

	$concatMap : (entry \to list') \to (list \to list')$
	$in : entry \times list \to bool$
	$exists, forall : (entry \to bool) \times list \to bool$
	$null : list \to bool$
`static preds`	$\_ \in \_ : entry \times list$
	$\_ \notin \_ : entry \times list$
	$\_ \not\equiv \_ : list \times list$
	$sorted : list(nat)$
`vars`	$x, y : entry \quad L, L' : list \quad L'' : list(list) \quad n : nat$
	$f : entry \to entry' \quad g : entry \to bool \quad h : entry \to list'$
`Horn axioms`	$eq(x, x) \equiv true$
	$eq(x, y) \equiv false \quad \Leftarrow \quad x \not\equiv y$
	$[x] \equiv x : []$
	$[] +\!\!+ L \equiv L$
	$(x : L) +\!\!+ L' \equiv x : (L +\!\!+ L')$
	$L`join`L' \equiv L +\!\!+ (L - L')$
	$L - L' \equiv filter(\lambda x.not(in(x, L')), L)$
	$[]!!n \equiv ()$
	$(x : L)!!0 \equiv (x)$
	$(x : L)!!(n+1) \equiv L!!n$
	$drop(n, []) \equiv []$
	$drop(0, L) \equiv L$
	$drop(n+1, x : L) \equiv drop(n, L)$
	$sublist([])(i, j) \equiv []$
	$sublist(x : L)(0, 0) \equiv []$
	$sublist(x : L)(0, j+1) \equiv x : sublist(L)(0, j)$
	$sublist(x : L)(i+1, j+1) \equiv sublist(L)(i, j)$
	$flatten([]) \equiv []$
	$flatten(L : L'') \equiv L +\!\!+ flatten(L'')$
	$mklist(()) \equiv []$
	$mklist((x_1, \ldots, x_n)) \equiv x_1 : mklist(x_2, \ldots, x_n)$
	$map(f)([]) \equiv []$
	$map(f)(x : L) \equiv f(x) : map(f)(L)$
	$filter(g, []) \equiv []$
	$filter(g, x : L) \equiv x : filter(g, L) \quad \Leftarrow \quad g(x) \equiv true$
	$filter(g, x : L) \equiv filter(g, L) \quad \Leftarrow \quad g(x) \equiv false$
	$length([]) \equiv 0$
	$length(x : L) \equiv length(L) + 1$
	$card([], x) \equiv 0$
	$card(x : L, x) \equiv card(L, x) + 1$
	$card(x : L, y) \equiv card(L, y) \quad \Leftarrow \quad x \not\equiv y$
	$concatMap(h)([]) \equiv []$
	$concatMap(h)(x : L) \equiv h(x) +\!\!+ concatMap(h)(L)$
	$in(x, L) \equiv exists(\lambda x.eq(x, y), L)$
	$exists(g, []) \equiv false$
	$exists(g, x : L) \equiv g(x)\ or\ exists(g, L)$
	$forall(g, []) \equiv true$
	$forall(g, x : L) \equiv g(x)\ and\ forall(g, L)$

$$null([]) \equiv true$$
$$null(L) \equiv false \quad \Leftarrow \quad L \not\equiv []$$
$$x \in L \quad \Leftarrow \quad in(x, L) \equiv true$$
$$x \notin L \quad \Leftarrow \quad in(x, L) \equiv false$$
$$sorted([])$$
$$sorted([x])$$
$$sorted(x : y : L) \quad \Leftarrow \quad x \leq y \wedge sorted(y : L)$$

standard inequality axioms (see [75], Section 4)

For any correct actualization SP of LIST that assigns the sort s to *entry*, $Ini(SP)_{list}$ is isomorphic to $Ini(SP)_s^*$ and thus to the initial F-algebra where $F(A) =_{def} 1 + (Ini(SP)_s \times A)$ (see Section 6).

A hidden specification of lists would amount to a specification of streams such as FSTREAM (cf. Section 4.2) where, however, all ground normal forms of sort *stream* denote *finite* streams.

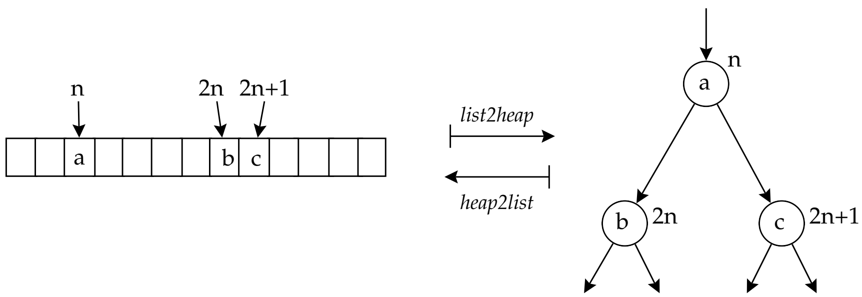

Figure 1. *From lists to heaps and backwards.*

The mutual translation between lists and binary trees according to the heap order is an instructive example of how to use sum sorts for specifying exceptions and *error recovery*.

HEAP = LIST **then**

hidsorts $bintree = bintree(entry)$

objconstructs $mt :\to bintree$

$\_\#\_\#\_ : bintree \times entry \times bintree \to bintree$

defuncts $list2heap : list \to bintree$

$mkHeap : list \times nat \to 1 + bintree$

$heap2list : bintree \to list$

$heaps2list : list(bintree) \times list \to list$

$root : bintree \to list$

$subtrees : bintree \to list(bintree)$

vars $x : entry \quad n : nat \quad L : list \quad T, T' : bintree \quad TL : list(bintree)$

Horn axioms

$$list2heap([]) \equiv mt$$
$$list2heap(x : L) \equiv T \quad \Leftarrow \quad mkHeap(x : L, 1) \equiv (T)$$
$$mkHeap(L, n) \equiv () \quad \Leftarrow \quad nth(L, n-1) \equiv ()$$
$$mkHeap(L, n) \equiv (mt\#x\#mt) \quad \Leftarrow \quad nth(L, n-1) \equiv (x) \wedge mkHeap(L, 2*n) \equiv () \wedge mkHeap(L, 2*n+1) \equiv ()$$
$$mkHeap(L, n) \equiv (mt\#x\#T) \quad \Leftarrow \quad nth(L, n-1) \equiv (x) \wedge mkHeap(L, 2*n) \equiv () \wedge mkHeap(L, 2*n+1) \equiv (T)$$
$$mkHeap(L, n) \equiv (T\#x\#mt) \quad \Leftarrow \quad nth(L, n-1) \equiv (x) \wedge$$

$$
\begin{array}{rcl}
 & & mkHeap(L, 2*n) \equiv (T) \wedge \\
 & & mkHeap(L, 2*n+1) \equiv () \\
mkHeap(L, n) \equiv (T\#x\#T') & \Leftarrow & nth(L, n-1) \equiv (x) \wedge \\
 & & mkHeap(L, 2*n) \equiv (T) \wedge \\
 & & mkHeap(L, 2*n+1) \equiv (T')
\end{array}
$$

$$
\begin{array}{l}
heap2list(T) \equiv heaps2list([T], []) \\
heaps2list([], L) \equiv L \\
heaps2list(T : TL, L) \\
\qquad \equiv \; heaps2list(concatMap(subtrees)(T : TL), L +\!\!+ concatMap(root)(T : TL)) \\
root(mt) \equiv [] \\
root(T\#x\#T') \equiv [x] \\
subtrees(mt) \equiv [] \\
subtrees(T\#x\#T') \equiv [T, T']
\end{array}
$$

Exception handling with sum sorts can be implemented directly in a functional language like ML as well as in a procedural language like Java by employing the corresponding language constructs (*raise* and *handle* in ML; *throw* and *try/catch* in Java).

1.2.2 Sets

Sets of a given set of entries can be specified in several ways. In the first version, many set operators are declared as constructors. All axioms are coinductive (cf. [78], Def. 5.1).

SET = LIST `then`

`hidsorts`	$set = set(entry)$	
`constructs`	$\emptyset, all :\to set$	
	$\{\_\} : entry \to set$	
	$\_ \cup \_ : set \times set \to set$	
	$\_ \setminus \_ : set \times set \to set$	
	$compr : (entry \to bool) \to set$	set comprehension
`destructs`	$in : entry \times set \to bool$	
`defuncts`	$\_ \cap \_ : set \times set \to set$	
	$filter : (entry \to bool) \times set \to set$	
	$insert, remove : entry \times set \to set$	
	$mkset : list \to set$	
	$mkset : entry^* \to set$	
`static preds`	$finite : set$	
	$\_ \in \_ : entry \times set$	
	$\_ \notin \_ : entry \times set$	
	$exists : (entry \to bool) \times set$	
ν`-preds`	$isempty, infinite : set$	
	$\_ \subseteq \_ : set \times set$	
	$forall : (entry \to bool) \times set$	
`vars`	$x, y : entry \quad s, s' : set \quad S : entry^* \quad L : list(entry) \quad g : entry \to bool$	
`Horn axioms`	$in(x, \emptyset) \equiv false$	
	$in(x, all) \equiv true$	
	$in(x, s \cup s') \equiv in(x, s) \; or \; in(x, s')$	
	$in(x, s \setminus s') \equiv in(x, s) \; and \; not(in(x, s'))$	

$$in(x, compr(g)) \equiv g(x)$$ $g(x)$ stands for $apply(g, x)$.
$$\{x\} \equiv compr(\lambda y.eq(x, y))$$
$$s \cap s' \equiv ((s \cup s') \setminus (s \setminus s')) \setminus (s' \setminus s)$$
$$filter(g, s) \equiv compr(g) \cap s$$
$$insert(x, s) \equiv s \cup \{x\}$$
$$remove(x, s) \equiv s \setminus \{x\}$$
$$mkset([]) \equiv \emptyset$$
$$mkset(x : L) \equiv insert(x, mkset(L))$$
$$mkset(S) \equiv mkset(mklist(S))$$
$$finite(s) \Leftarrow isempty(s)$$
$$finite(s) \Leftarrow x \in s \wedge finite(remove(x, s))$$
$$x \in s \Leftarrow in(x, s) \equiv true$$
$$x \notin s \Leftarrow in(x, s) \equiv false$$
$$exists(g, s) \Leftarrow x \in s \wedge g(x) \equiv true$$

`co-Horn axioms`
$$isempty(s) \Rightarrow (x \in s \Rightarrow False)$$
$$infinite(s) \Rightarrow (finite(s) \Rightarrow False)$$
$$s \subseteq s' \Rightarrow (x \in s \Rightarrow x \in s')$$
$$forall(g, s) \Rightarrow (x \in s \Rightarrow g(x) \equiv true)$$

Regardless of the actualization of *entry*, as a non-coalgebraic swinging type, SET presents only those sets of entries that are reachable by the respective set constructors, i.e., only countable sets. If SET is regarded as a coalgebraic swinging type, SET presents all elements even of an uncountable powerset.

For specifying defined functions whose axioms use ν-predicates we regard SET as the visible subspecification of the following extension of SET:

SET1 = SET `and` NAT_∞ `then`

`defuncts`
$$map : (entry \to entry') \times set \to set$$
$$exists : (entry \to bool) \times set \to bool$$
$$|\_| : set \to nat$$

`vars` $x : entry \quad s : set \quad f : entry \to entry' \quad g : entry \to bool$

`Horn axioms`
$$exists(g, s) \equiv true \Leftarrow exists(g, s)$$
$$exists(g, s) \equiv false \Leftarrow forall(not \circ g, s)$$
$$map(f, s) \equiv compr(exists(\lambda y.eq(f(y), x), s))$$
$$|s| \equiv 0 \Leftarrow isempty(s)$$
$$|s| \equiv |remove(x, s)| + 1 \Leftarrow finite(s) \wedge x \in s$$
$$|s| \equiv \infty \Leftarrow infinite(s)$$

Some axioms might look rather unusual. The reason is that they should be coinductive. Coinductivity forbids axioms for defined functions or μ-predicates that represent inductive definitions on *hidden* normal forms. However, condition (3) of the definition of a coinductive specification ([78], 5.1) also admits non-coinductive axioms for a function or predicate whose compatibility with behavioral equivalence can be shown directly. For instance, if we restrict ourselves to a specification of *finite* sets, the constructors $\emptyset$, $\{\_\}$ and $\cup$ are sufficient for building up all members of the set domain so that the following specification is complete — and coinductive because it can be proved directly that each symbol whose axioms are not coinductive is compatible with behavioral equivalence.

FINSET = LIST `then`

`hidsorts` $set = set(entry) \quad set' = set(entry')$

`constructs`	$\emptyset :\to set$
	$\{\_\} : entry \to set$
	$\_ \cup \_ : set \times set \to set$
`destructs`	$in : entry \times set \to bool$
`defuncts`	$\_ \cap \_ : set \times set \to set$
	$insert, remove : entry \times set \to set$
	$mkset : list \to set$
	$mkset : entry^* \to set$
	$\_ \setminus \_ : set \times set \to set$
	$filter : (entry \to bool) \times set \to set$
	$map : (entry \to entry') \times set \to set'$
	$\lvert \_ \rvert : set \to nat$
	$exists, forall : (entry \to bool) \times set \to bool$
	$isempty : set \to bool$
	$flatten : set(set) \to set$
`static preds`	$\_ \in \_ : entry \times set$
	$\_ \notin \_ : entry \times set$
	$exists, forall : (entry \to bool) \times set$
	$isempty : set$
	$\_ \subseteq \_ : set \times set$
`vars`	$x, y, x_1, \ldots, x_n : entry \quad s, s', s'' : set \quad f : entry \to entry' \quad g : entry \to bool \quad L : list$
`Horn axioms`	$in(x, \emptyset) \;\equiv\; false$
	$in(x, \{y\}) \;\equiv\; eq(x, y)$
	$in(x, s \cup s') \;\equiv\; in(x, s) \text{ or } in(x, s')$
	$s \cap s' \;\equiv\; ((s \cup s') \setminus (s \setminus s')) \setminus (s' \setminus s)$
	$insert(x, s) \;\equiv\; s \cup \{x\}$
	$remove(x, s) \;\equiv\; s \setminus \{x\}$
	$mkset([]) \;\equiv\; \emptyset$
	$mkset(x : L) \;\equiv\; insert(x, mkset(L))$
	$mkset(()) \;\equiv\; \emptyset$
	$mkset((x_1, \ldots, x_n)) \;\equiv\; \{x_1\} \cup \cdots \cup \{x_n\}$
	$s \setminus s' \;\equiv\; filter(\lambda x.not(in(x, s')), s)$
	$filter(g, \emptyset) \;\equiv\; \emptyset$
	$filter(g, \{x\}) \;\equiv\; \emptyset \quad \Leftarrow \quad g(x) \equiv false$
	$filter(g, \{x\}) \;\equiv\; \{x\} \quad \Leftarrow \quad g(x) \equiv true$
	$filter(g, s \cup s') \;\equiv\; filter(g, s) \cup filter(g, s')$
	$map(f, \emptyset) \;\equiv\; \emptyset$
	$map(f, \{x\}) \;\equiv\; \{f(x)\}$
	$map(f, s \cup s') \;\equiv\; map(f, s) \cup map(f, s')$
	$\lvert\emptyset\rvert \;\equiv\; 0$
	$\lvert\{x\}\rvert \;\equiv\; 1$
	$\lvert s \cup s'\rvert \;\equiv\; \lvert s \setminus s'\rvert + \lvert s' \setminus s\rvert$
	$exists(g, \emptyset) \;\equiv\; false$
	$exists(g, \{x\}) \;\equiv\; g(x)$
	$exists(g, s \cup s') \;\equiv\; exists(g, s) \text{ or } exists(g, s')$
	$forall(g, s) \;\equiv\; not(exists(not \circ g, s))$
	$isempty(\emptyset) \;\equiv\; true$
	$isempty(s \cup s') \;\equiv\; isempty(s) \text{ and } isempty(s')$

$$
\begin{array}{l}
flatten(\emptyset) \equiv \emptyset \\
flatten(\{s\}) \equiv s \\
flatten(s \cup s') \equiv flatten(s) \cup flatten(s') \\
x \in s \Leftarrow in(x,s) \equiv true \\
x \notin s \Leftarrow in(x,s) \equiv false \\
exists(g,s) \Leftarrow exists(g,s) \equiv true \\
forall(g,s) \Leftarrow forall(g,s) \equiv true \\
\emptyset \subseteq s \\
\{x\} \subseteq s \Leftarrow x \in s \\
(s \cup s') \subseteq s'' \Leftarrow s \subseteq s'' \wedge s' \subseteq s''
\end{array}
$$

The following specification, LIST2SET, of finite sets actually presents lists. For obtaining sets, we apply a model transformer to the final model of LIST2SET. Given a signature Σ and a subsignature Σ' of Σ, the **restriction operator induced by** Σ', $reach_{\Sigma'}$, maps a Σ-structure A to the least Σ'-substructure A' contained in A. For all sorts $s \in \Sigma'$, A'_s consists of all elements of A that have a ground Σ'-term representation, i.e. $a = t^A$ for some $t \in T_{\Sigma'}$. Given that DF is the set of defined functions of a swinging type SP and EF is the set of those defined functions that are declared as **generators**, the standard model of SP is defined as $reach_{\Sigma \setminus DF \cup EF}(Fin(SP))$.

If SP=LIST2SET, this model consists of all sorted lists of entries, which provide unique representations of finite sets of entries

ORDER(*entry*) = ENTRY(*entry*) `then`

`preds`	$\_ \leq \_ : entry \times entry$
	$\_ > \_ : entry \times entry$
	$\_ < \_ : entry \times entry$
`vars`	$x, y : entry$
`axioms`	$x \leq y \dot{\vee} x > y$ $x < y \Leftrightarrow y > x$ $x \leq y \Leftrightarrow (x < y \dot{\vee} x \equiv y)$

LIST2SET = ORDER(*entry*) `and` ORDER(*entry*′) `and` NAT `then`

`hidsorts`	$set = set(entry)$ $set' = set(entry')$
`objconstructs`	$[] : \rightarrow set$
	$\_ : \_ : entry \times set \rightarrow set$
`generators`	$\emptyset : \rightarrow set$
	$\{\_\} : entry \rightarrow set$
	$\cup : set \times set \rightarrow set$
	$insert, remove : entry \times set \rightarrow set$
	$mkset : list \rightarrow set$
	$filter : (entry \rightarrow bool) \times set \rightarrow set$
	$map : (entry \rightarrow entry') \times set \rightarrow set'$
	$\lvert\_\rvert : set \rightarrow nat$
	$in : entry \times set \rightarrow bool$
	$exists, forall : (entry \rightarrow bool) \times set \rightarrow bool$
	$isempty : set \rightarrow bool$
	$flatten : set(set) \rightarrow set$
`static preds`	$\_ \in \_ : entry \times list$
	$\_ \notin \_ : entry \times list$
	$\_ \subseteq \_ : set \times set$
`vars`	$x, y : entry$ $s, s' : set$ $S : set(set)$ $f : entry \rightarrow entry'$ $g : entry \rightarrow bool$

	$L : list$
`Horn axioms`	$\emptyset \equiv []$
	$\{x\} \equiv x : []$
	$[] \cup s \equiv s$
	$(x : s) \cup s' \equiv insert(x, s \cup s')$
	$insert(x, []) \equiv x : []$
	$insert(x, y : s) \equiv x : s$
	$insert(x, y : s) \equiv x : (y : s) \quad \Leftarrow \quad x < y$
	$insert(x, y : s) \equiv y : insert(x, s) \quad \Leftarrow \quad x > y$
	$remove(x, s) \equiv filter(\lambda y.not(eq(x, y)), s)$
	$mkset([]) \equiv \emptyset$
	$mkset(x : L) \equiv insert(x, mkset(L))$
	$filter(g, []) \equiv []$
	$filter(g, x : s) \equiv filter(g, s) \quad \Leftarrow \quad g(x) \equiv false$
	$filter(g, x : s) \equiv x : filter(g, s) \quad \Leftarrow \quad g(x) \equiv true$
	$map(f, []) \equiv []$
	$map(f, x : s) \equiv insert(f(x), map(f, s))$
	$\lvert[]\rvert \equiv 0$
	$\lvert x : s\rvert \equiv \lvert s\rvert + 1$
	$in(x, s) \equiv exists(\lambda y.eq(x, y), s)$
	$exists(g, []) \equiv false$
	$exists(g, x : s) \equiv (g(x)\ or\ exists(g, s)$
	$forall(g, s) \equiv not(exists(not \circ g, s))$
	$isempty([]) \equiv true$
	$isempty(x : s) \equiv false$
	$flatten([]) \equiv []$
	$flatten(s : S) \equiv s \cup flatten(S)$
	$x \in s \quad \Leftarrow \quad in(x, s) \equiv true$
	$x \notin s \quad \Leftarrow \quad in(x, s) \equiv false$
	$[] \subseteq s$
	$x : s \subseteq s' \quad \Leftarrow \quad x \in s' \wedge s \subseteq s'$

Since the generators create only sorted lists or transform sorted lists into sorted lists, each ground term built up of generators is structurally equivalent to a normal form representing a sorted list. Hence each finite set has a unique representation in $reach_{\Sigma \setminus DF \cup EF}(Fin(\text{LIST2SET}))$.

1.2.3 Multisets

The behavioral equivalence of bags or multisets is determined by the destructor *card*, which returns the number of occurrences of a given entry in a bag. Two finite bags are behaviorally equivalent iff the lists they are constructed from are permutations of each other. Hence a specification of bags can be used, for instance, for proving conjectures about list permutations, such as the condition that a sorting algorithm returns a permutation of its input.

BAG = LIST `then`

`hidsorts`	$bag = bag(entry) \quad bag' = bag(entry')$
`constructs`	$empty :\to bag$
	$[\_] : entry \to bag$

	$\_+\_ : bag \times bag \rightarrow bag$
	$\_-\_ : bag \times bag \rightarrow bag$
	$map : (entry \rightarrow entry') \times bag \rightarrow bag'$
`destructs`	$card : bag \times entry \rightarrow nat$
`defuncts`	$mkbag : list \rightarrow bag$
`static preds`	$\_\in\_ : entry \times bag$
	$exists : (entry \rightarrow bool) \times set$
ν`-preds`	$isempty : bag \times bag$
	$\_\subseteq\_ : bag \times bag$
	$forall : (entry \rightarrow bool) \times set$
`vars`	$x, y : entry \quad b, b' : bag \quad f : entry \rightarrow entry' \quad g : entry \rightarrow bool \quad L : list$
`Horn axioms`	$card([x], x) \equiv 1$
	$card([x], y) \equiv 0 \Leftarrow x \not\equiv y$
	$card(b + b', x) \equiv card(b, x) + card(b', x)$
	$card(b - b', x) \equiv card(b, x) - card(b', x)$
	$card(empty, x) \equiv 0$
	$card(map(f, b), x) \equiv card(b, y) \Leftarrow f(y) \equiv x$
	$mkbag([]) \equiv empty$
	$mkbag(x : L) \equiv [x] + mkbag(L)$
	$x \in b \Leftarrow card(b, x) > 0$
	$exists(g, b) \Leftarrow x \in b \wedge g(x) \equiv true$
`co-Horn axioms`	$isempty(b) \Rightarrow (x \in b \Rightarrow False)$
	$b \subseteq c \Rightarrow (x \in b \Rightarrow x \in c)$
	$forall(g, b) \Rightarrow (x \in b \Rightarrow g(x) \equiv true)$

All axioms for bag functions are coinductive except those for $|\_|$. Similarly to the step from SET to FINSET, we may turn several bag constructors into defined functions if only finite bags are to be specified. Again, many of the new axioms are not coinductive. Hence the compatibility of behavioral equivalence with defined functions must be proved explicitly. For minimizing the number of cases to be considered we reduce the set of constructors to a singleton, namely $mkbag : list \rightarrow bag$. Behavioral FINBAG-equivalence actually coincides with the equivalence kernel of (the interpretation of) $mkbag$ (in the Herbrand FINBAG-model).

FINBAG = LIST `then`

`hidsorts`	$bag = bag(entry) \quad bag' = bag(entry')$
`constructs`	$mkbag : list \rightarrow bag$
`destructs`	$card : bag \times entry \rightarrow nat$
`defuncts`	$empty : \rightarrow bag$
	$[\_] : entry \rightarrow bag$
	$\_+\_ : bag \times bag \rightarrow bag$
	$\_-\_ : bag \times bag \rightarrow bag$
	$map : (entry \rightarrow entry') \times bag \rightarrow bag'$
	$filter : (entry \rightarrow bool) \times bag \rightarrow bag$
	$exists : (entry \rightarrow bool) \times bag \rightarrow bool$
	$forall : (entry \rightarrow bool) \times bag \rightarrow bool$
	$isempty : bag \rightarrow bool$
	$\vert\_\vert : bag \rightarrow nat$
`static preds`	$\_\in\_ : entry \times bag$
ν`-preds`	$\_\subseteq\_ : bag \times bag$

`vars`	$x, y : entry \quad b, b' : bag \quad f : entry \to entry' \quad g : entry \to bool \quad L : list$
`Horn axioms`	$card(mkbag(L), x) \ \equiv\ card(L, x)$
	$empty \equiv mkbag([])$
	$[x] \ \equiv\ mkbag([x])$
	$mkbag(L) + mkbag(L') \ \equiv\ mkbag(L +\!\!+ L')$
	$mkbag(L) - mkbag(L') \ \equiv\ mkbag(L - L')$
	$map(f, mkbag(L)) \ \equiv\ mkbag(map(f)(L))$
	$filter(g, mkbag(L)) \ \equiv\ mkbag(filter(g, L))$
	$exists(g, mkbag(L)) \ \equiv\ exists(g, L)$
	$forall(g, mkbag(L)) \ \equiv\ forall(g, L)$
	$isempty(mkbag(L)) \ \equiv\ null(L)$
	$\lvert mkbag(L)\rvert \ \equiv\ length(L)$
	$x \in mkbag(L) \quad \Leftarrow \quad x \in L$
`co-Horn axioms`	$b \subseteq c \quad \Rightarrow \quad (x \in b \ \Rightarrow\ x \in c)$

1.2.4 Maps

The third common schema of a *permutative* type provides partial functions, also called **arrays** if the doamin is finite, or **tables** or **matrices** if the domain is a binary relation.

MAP = ENTRY(*domain*) `and` ENTRY(*range*) `and` SET `then`

`hidsorts`	$map = map(domain, range)$	
`constructs`	$new : \to map$	
	$upd : domain \times range \times map \to map$	update
`destructs`	$get : map \times domain \to 1 + range$	
`defuncts`	$dom : map \to set(domain)$	
	$ran : map \to set(range)$	
	$pre : map \times range \to set(domain)$	
	$remove : domain \times map \to map$	
	$\_ * \_ : map \times (range \to range) \to map$	
`vars`	$i, j : domain \quad x, y : range \quad f : map \quad h : range \to range$	
`Horn axioms`	$get(new, i) \equiv ()$	
	$get(upd(i, x, f), i) \equiv (x)$	
	$get(upd(i, x, f), j) \equiv get(f, j) \quad \Leftarrow \quad i \not\equiv j$	
	$dom(new) \equiv \emptyset$	
	$dom(upd(i, x, f)) \equiv insert(i, dom(f))$	
	$ran(new) \equiv \emptyset$	
	$ran(upd(i, x, f)) \equiv insert(x, ran(f))$	
	$pre(new, x) \equiv \emptyset$	
	$pre(upd(i, x, f), x) \equiv insert(i, pre(f, x))$	
	$pre(upd(i, x, f), y) \equiv pre(f, y) \quad \Leftarrow \quad x \not\equiv y$	
	$remove(i, new) \equiv new$	
	$remove(i, upd(i, x, f)) \equiv remove(i, f)$	
	$remove(i, upd(j, x, f)) \equiv upd(j, x, remove(i, f)) \quad \Leftarrow \quad i \not\equiv j$	
	$new * h \equiv new$	
	$upd(i, x, f) * h \equiv upd(i, h(x), f * h)$	

MAP+ = MAP `then`

`defuncts`	$\_+\_ : map \times map \rightarrow 1 + map$
`vars`	$i : domain \quad x : range \quad f, g, f' : map$
`Horn axioms`	$new + f \equiv (f)$
	$upd(i,x,f) + g \equiv (upd(i,x,f')) \quad \Leftarrow \quad i \notin dom(g) \wedge f + g \equiv (f')$
	$upd(i,x,f) + g \equiv () \quad \Leftarrow \quad i \notin dom(g) \wedge f + g \equiv ()$
	$upd(i,x,f) + g \equiv () \quad \Leftarrow \quad i \in dom(g)$

The following function *uses* adopted from [83] transforms a list L of map-updates and -lookups into the list of values returned by the lookups of L:

USELIST = MAP `and` LIST `then`

`vissorts`	occ
`constructs`	$def : domain \times range \rightarrow occ$
	$use : domain \rightarrow occ$
`defuncts`	$uses : list(occ) \rightarrow list(1 + range)$
	$loop : list(occ) \times map \rightarrow list(1 + range)$
`vars`	$L : list(occ) \quad i : domain \quad x : range \quad f : map$
`Horn axioms`	$uses(L) \equiv loop(L, new)$
	$loop([], f) \equiv []$
	$loop(def(i,x) : L, f) \equiv loop(L, upd(i,x,f))$
	$loop(use(i) : L, f) \equiv get(f,i) : loop(L,f)$

An implementation of USELIST that regards the variable f only as a pointer to an object of sort *map* would not comply with the intended semantics of the specification that is given by the Herbrand model of USELIST. The error occurs when the last axiom is applied and f is copied. For instance, consider the following reduction:

$$uses([def(1,a), use(1), def(1,b)]) \longrightarrow loop([def(1,a), use(1), def(1,b)], new)$$
$$\longrightarrow loop([use(1), def(1,b)], upd(1,a,new)) \longrightarrow get(\underline{upd(1,a,new)}, 1) : loop([def(1,b)], \underline{upd(1,a,new)}).$$

If the last step is implemented in a *multi-threaded* way, i.e. instead of copying $upd(1,a,new)$ a second reference to this term is generated, then the subsequent reduction of $loop([def(1,b)], upd(1,a,new))$, which leads to the replacement of $upd(1,a,new)$ by $upd(1,b,upd(1,a,new))$, implicitly rewrites $get(upd(1,a,new),1)$ to the inequivalent term $get(upd(1,b,upd(1,a,new)),1)$.

If the domain is ordered, we may specialize MAP to BMAP ("bounded maps") such that the final BMAP-model identifies all maps with identical restrictions to a given interval of domain elements.

BMAP = ORDER(*domain*) `and` ENTRY(*range*) `then`

`hidsorts`	$bmap = bmap(domain, range)$
`constructs`	$new : domain \times domain \rightarrow bmap$
	$upd : domain \times range \times bmap \rightarrow bmap$
`destructs`	$get : bmap \times domain \rightarrow 1 + range$
`defuncts`	$lwb, upb : bmap \rightarrow domain$
`vars`	$i, j, k : domain \quad x : range \quad f : bmap$
`Horn axioms`	$get(new(i,j), k) \equiv ()$
	$get(upd(i,x,f), i) \equiv (x) \quad \Leftarrow \quad lwb(f) \leq i \wedge i \leq upb(f)$
	$get(upd(i,x,f), i) \equiv () \quad \Leftarrow \quad lwb(f) > upb(f)$
	$get(upd(i,x,f), i) \equiv () \quad \Leftarrow \quad lwb(f) > i$
	$get(upd(i,x,f), i) \equiv () \quad \Leftarrow \quad i > upb(f)$
	$get(upd(i,x,f), j) \equiv get(f,j) \quad \Leftarrow \quad i \not\equiv j$

$$lwb(new(i,j)) \equiv i \qquad upb(new(i,j)) \equiv j$$
$$lwb(upd(i,x,f)) \equiv lwb(f) \qquad upb(upd(i,x,f)) \equiv upb(f)$$

1.2.5 Monads

Monads, **Kleisli triples** or **algebraic theories** [61, 69, 83, 101] are parameterized domains $M(s)$ such that, category-theoretically, M is an endofunctor and, intuitively, M stands for a notion of computation, while $M(s)$ denotes the set of M-computations of values of sort s. Given a set S of sorts and a category $\mathcal{K}$, suppose that each $s \in S$ denotes an object of $\mathcal{K}$. A function $M : Obj(\mathcal{K}) \to Obj(\mathcal{K})$ on a category $\mathcal{K}$ is a monad if for each $s \in S$ there is a function $unit = unit_s : s \to M(s)$ and for each function $f : s \to M(s')$ there is a function $f^* : M(s) \to M(s')$ such that $unit_s^*$ is the identity on $M(s)$ and for all $f : s \to M(s')$ and $g : s' \to M(s'')$, $f^* \circ unit_s = f$ and $g^* \circ f^* = (g^* \circ f)^*$. Intuitively, $unit_s$ embeds s into $M(s)$ and f^* extends f from s to $M(s)$. M becomes an endofunctor on $\mathcal{K}$ by defining $M(f : s \to s')$ as $(unit_{s'} \circ f)^*$. The list monad is a built-in monad of Haskell.

Common monads are the list or free-monoid functor $\_^* : Set^S \to Set^S$ with $unit(a) =_{def} [a]$ and $f^*(L) =_{def} concatMap(f)(L)$, the powerset functor $\wp : Set^S \to Set^S$ with $unit(a) =_{def} \{a\}$ and $f^*(A) =_{def} \cup_{a \in A} f(a)$, and for a signature $\Sigma = (S, F)$, the term or free-algebra functor $T_\Sigma : Set^S \to Set^S$ with $unit(a) =_{def} a$ and $f^*(t(a_1, \dots, a_n)) =_{def} t(f(a_1), \dots, f(a_n))$.

Given a sort s, the **state monad** $M(s)$ is a functional sort of the form $[state \to (s \times state)]$ that denotes a set of state transformations with outputs in s. For all S-sorted sets A and $s \in S$, $A_{M(s)} =_{def} [Q \to (A_s \times Q)]$.

STATEMON = ENTRY($state$) `and` ENTRY(s) `and` ENTRY(s') `then`

`hidsorts`	$M(s) \;=\; state \to (s \times state)$	
`destructs`	$apply : M(s) \times state \to s \times state$	
	$apply : (s \to M(s')) \times s \to M(s')$	
`defuncts`	$return : s \to M(s)$	Haskell notation for $unit$
	$\_^* : (s \to M(s')) \to (M(s) \to M(s'))$	
	$\_ \gg= \_ : M(s) \times (s \to M(s')) \to M(s')$	
	$\_ \gg \_ : M(s) \times M(s') \to M(s')$	
	$\_ \circ \_ : (s' \to M(s'')) \times (s \to M(s')) \to (s \to M(s''))$	
`vars`	$x : s \quad st, st' : state \quad m : M(s) \quad m' : M(s') \quad f : s \to M(s') \quad g : s' \to M(s'')$	
`Horn axioms`	$return(x)(st) \equiv (x, st)$	
	$f^*(m)(st) \equiv f(x)(st') \;\Leftarrow\; m(st) \equiv (x, st')$	
	$m \gg= f \;\equiv\; f^*(m)$	
	$m \gg m' \;\equiv\; m \gg= \lambda x.m'$	
	$(g \circ f)(x) \equiv f(x) \gg= g$	

Parser monads [11] are of the form

$$M(tree) \;=\; input \to list(tree \times input)$$

where $input$ is usually a product of string sorts and $tree$ is a sort for derivation trees. For an actual input xs, $M(tree)(xs)$ is a list of parses of xs each of which consists of a derivation tree for some prefix of xs and the remaining suffix of xs.

The composition operators $\gg=$, $\gg$ and $\circ$ have the same axioms in all monads.[1] $\gg=$ usually occurs in a context of the form $m \gg= \lambda x.m'$. If one unrolls the semantics of $\gg=$, as it is given by the axioms of

[1] $\gg=$ and $\gg$ are the notations used in the functional programming language Haskell [45], whose imperative features are based on built-in monads.

STATEMON, $m \ggg= \lambda x.m'$ turns out to represent an assignment to x of the output of the state transformation m, followed by the state transformation m' that uses (the assigned value of) x. This motivates Haskell's *do* notation [11, 45] for nested bind expressions:

$$m_1 \ggg= \lambda x_1.(m_2 \ggg= \lambda x_2.(\dots(m_n \ggg= \lambda x_n.m)\dots))$$

is denoted by

$$do\{x_1 \leftarrow m_1;\ x_2 \leftarrow m_2;\ \dots;\ x_n \leftarrow m_n;\ m\}.$$

A recursive compiler of *do*-expressions into bind expressions is defined as follows:

$$\begin{aligned} comp(do\{m\}) &= m \\ comp(do\{m;\ R\}) &= m \gg comp(do\{R\}) \\ comp(do\{x \leftarrow m;\ R\}) &= m \ggg= \lambda x.comp(do\{R\}) \end{aligned}$$

With the help of STATEMON functions generating or modifying states are turned into procedures, i.e. their axioms look more like imperative than functional-logic programs. This admits the communication between programs of different types, in particular, at places where I/O is performed. Therefore, Haskell ([45]) and Curry ([38]) employ a built-in I/O monad whose states are hidden insofar as they can only be accessed indirectly via monad functions. Hence the state is *single-threaded*, i.e. always referenced by at most one pointer and thus "destructive updates" of the store are safe. If a program based on STATEMON does not use state-sorted terms, states can never be copied like, for instance, the map f is copied in the last axiom of USELIST (cf. Section 1.2.4).

If the state sort of STATEMON is actualized by *map*, one comes up with the *monadic arrays* of [83]:

MAPMON = MAP `and` STATEMON `then`

`hidsorts`	$map = map(domain, range) \quad M(s) = map \rightarrow (s \times map)$
`defuncts`	$New : M(s) \rightarrow s$
	$Upd : domain \times range \rightarrow M(1)$
	$Get : domain \rightarrow M(1 + range)$
`vars`	$i : domain \quad x : range \quad y : s \quad m : M(s) \quad f : map$
`Horn axioms`	$New(m) \equiv y \quad \Leftarrow \quad m(new) \equiv (y, f)$
	$Upd(i, x)(f) \equiv ((), upd(i, x, f))$
	$Get(i)(f) \equiv (get(f, i), f)$

Following [83] we re-program USELIST (cf. 1.2.4) in terms of MAPMON whereby maps become single-threaded:

USELIST = MAPMON `and` LIST `then`

`sorts`	occ
`constructs`	$def : domain \times range \rightarrow occ$
	$use : domain \rightarrow occ$
`defuncts`	$uses : list(occ) \rightarrow list(1 + range)$
	$Loop : list(occ) \rightarrow M(list(1 + range))$
`vars`	$L : list(occ) \quad i : domain \quad x : range \quad x' : 1 + range \quad L' : list(1 + range)$
`Horn axioms`	$uses(L) \equiv New(Loop(L))$
	$Loop([]) \equiv return([])$
	$Loop(def(i, x) : L) \ \equiv\ do\{Upd(i, x);\ Loop(L)\}$
	$Loop(use(i) : L) \ \equiv\ do\{x' \leftarrow Get(i);\ L' \leftarrow Loop(L);\ return(x' : L')\}$

From the monad version of USELIST we directly obtain an imperative program. We remove the monad constructor M, get the type of *Loop* as a *procedure*:

$$Loop : list(occ) \to list(1 + range),$$

introduce a state variable $f : map$, remove the monad embedding *return* and translate the axioms of USELIST into Java method declarations:

$$\begin{array}{l} list(1+range)\ uses(list(occ)\ L)\ \{f := new;\ return\ Loop(L)\} \\ list(1+range)\ Loop(list(occ)\ L)\ \{switch\ L\ \{\quad case\ []\ :\ return\ [];\\ \qquad\qquad case\ def(i,x):L_1)\ :\ Upd(i,x);\ return\ Loop(L_1);\\ \qquad\qquad case\ Loop(use(i):L_1)\ :\\ \qquad\qquad\qquad x' := Get(i);\ L' := Loop(L_1);\ return\ x' : L'\}\} \end{array}$$

The sort $1 + map$ of MAP is a sum sort that was introduced in order to totalize partial functions with range sort *map* (cf. Section 1.2.4). Hence $1 + map$ is derived from an **exception** or **error monad** [69, 101]:

ERRORMON = ENTRY(s) `then`

`sorts`	$E(s) = 1 + s$
`defuncts`	$unit : s \to E(s)$
	$\_^* : (s \to E(s')) \to (E(s) \to E(s'))$
	$\_ \rhd \_ : E(s) \times (s \to E(s')) \to E(s')$
`vars`	$x : s \quad e : E(s) \quad f : s \to E(s')$
`Horn axioms`	$unit(x) \equiv (x)$
	$f^*((x)) \equiv f(x)$
	$f^*(()) \equiv ()$
	$e \rhd f \equiv f^*(e)$

The axioms for $+ : map \times map \to 1 + map$ (cf. 1.2.4) follow a schema that becomes obvious if we respecify this function with the help of an error monad:

MAP++ = MAP `and` ERRORMON `then`

`defuncts`	$\_ + \_ : map \times map \to E(map)$
`vars`	$i : domain \quad x : range \quad f, g, f' : map$
`Horn axioms`	$new + f \equiv (f)$
	$upd(i,x,f) + g \equiv (f+g) \rhd \lambda f'.(upd(i,x,f')) \Leftarrow i \notin dom(g)$
	$upd(i,x,f) + g \equiv () \Leftarrow i \in dom(g)$

2 Trees, graphs, and parsers

2.1 Regular trees

A tree or rooted graph is **regular** if all nodes are reachable from a root node and nodes with the same label have the same outdegree.

Given that the number of different node labels is finite, a finite regular tree can be specified as a ground normal form. Each node label becomes a constructor whose arity agrees with the node's outdegree.

REGTREE

`sorts`	$tree$	
`constructs`	$c_1, \ldots, c_m :\rightarrow tree$	
	$f_1 : tree^{k_1} \rightarrow tree$	$k_1 > 0$
	$\ldots$	
	$f_n : tree^{k_n} \rightarrow tree$	$k_n > 0$
`defuncts`	$subs : tree \rightarrow \amalg_{i=1}^{m} 1 + \amalg_{i=1}^{n} tree^{k_i}$	subtrees
`vars`	$T_1, \ldots, T_{k_i} : tree$	$1 \leq i \leq n$
`Horn axioms`	$subs(c_i) \equiv \kappa_i()$	
	$subs(f_i(T_1, \ldots, T_{k_i})) \equiv \kappa_{m+i}(T_1, \ldots, T_{k_i})$	

As an example, let us specify **binary decision trees** that are used for representing n-ary Boolean functions and manipulating the representations:

BDTREE = BOOL `and` FINSET `then`

`sorts`	$bdtree$	
`constructs`	$0, 1 :\rightarrow bdtree$	
	$f_1, \ldots, f_n : bdtree \times bdtree \rightarrow bdtree$	
`defuncts`	$subs : bdtree \rightarrow 1 + 1 + \amalg_{i=1}^{n}(bdtree \times bdtree)$	
	$tree2fun : bdtree \rightarrow (bool^n \rightarrow bool)$	
	$\neg : bdtree \rightarrow bdtree$	
	$+ : bdtree \times bdtree \rightarrow bdtree$	
	$restrict_i : bdtree \times bool \rightarrow bdtree$	$1 \leq i \leq n$
	$reduce : bdtree \rightarrow bdtree$	
	$solve : bdtree \rightarrow 1 + bool^n$	
	$Solve : bdtree \rightarrow set(bool^n)$	
	$all :\rightarrow set(bool^n)$	
	$compose : bdtree \times bdtree \rightarrow bdtree$	
	$glue : bdtree \times bdtree \times bdtree \rightarrow bdtree$	
`vars`	$T, T', T_1, T_2 : bdtree \quad x, x_1, \ldots, x_n : bool$	
`Horn axioms`	$subs(0) \equiv \kappa_1()$	
	$subs(1) \equiv \kappa_2()$	
	$subs(f_i(T, T')) \equiv \kappa_{2+i}(T, T')$	$1 \leq i \leq n$
	$tree2fun(0) \equiv \lambda(x_1, \ldots, x_n).false$	
	$tree2fun(1) \equiv \lambda(x_1, \ldots, x_n).true$	
	$tree2fun(f_i(T, T')) \equiv \lambda(x_1, \ldots, x_n).((not(x_i)\ and\ tree2fun(T))\ or$ $(x_i\ and\ tree2fun(T')))$	
	$\neg 0 \equiv 1$	
	$\neg 1 \equiv 0$	
	$\neg f_i(T, T') \equiv f_i(\neg T, \neg T')$	
	$0 + T \equiv T$	
	$1 + T \equiv 1$	
	$T + 0 \equiv T$	
	$T + 1 \equiv 1$	
	$f_i(T, T') + f_i(T_1, T_2) \equiv f_i(T + T_1, T' + T_2)$	
	$f_i(T, T') + f_j(T_1, T_2) \equiv f_i(T + f_j(T_1, T_2), T' + f_j(T_1, T_2))$	$1 \leq i < j \leq n$
	$f_i(T, T') + f_j(T_1, T_2) \equiv f_j(T_1 + f_i(T, T'), T_2 + f_i(T, T'))$	$1 \leq j < i \leq n$
	$restrict_i(0, x) \equiv 0$	
	$restrict_i(1, x) \equiv 1$	

$$
\begin{array}{ll}
restrict_i(f_i(T,T'),false) \equiv T & \\
restrict_i(f_i(T,T'),true) \equiv T' & \\
restrict_i(f_j(T,T'),x) \equiv f_j(restrict_i(T,x),restrict_i(T',x)) & 1 \leq i \neq j \leq n \\
reduce(0) \equiv 0 & \\
reduce(1) \equiv 1 & \\
reduce(f_i(T,T')) \equiv T_1 \quad \Leftarrow \quad reduce(T) \equiv T_1 \wedge reduce(T') \equiv T_2 \wedge T_1 \equiv T_2 & \\
reduce(f_i(T,T')) \equiv f_i(T_1,T_2) \quad \Leftarrow \quad reduce(T) \equiv T_1 \wedge reduce(T') \equiv T_2 \wedge T_1 \not\equiv T_2 & \\
solve(0) \equiv () & \\
solve(1) \equiv (true,\dots,true) & \\
solve(f_i(T,T')) \equiv (x_1,\dots,x_{i-1},false,x_{i+1},\dots,x_n) \quad \Leftarrow \quad solve(T) \equiv (x_1,\dots,x_n) & \\
solve(f_i(T,T')) \equiv (x_1,\dots,x_{i-1},true,x_{i+1},\dots,x_n) \quad \Leftarrow \quad solve(T') \equiv (x_1,\dots,x_n) & \\
solve(f_i(T,T')) \equiv () \quad \Leftarrow \quad solve(T) \equiv () \wedge solve(T') \equiv () & \\
Solve(0) \equiv \emptyset & \\
Solve(1) \equiv all & \\
Solve(f_i(T,T')) \equiv filter(\lambda(x_1,\dots,x_n).eq(x_i,false),Solve(T)) \cup & \\
\qquad filter(\lambda(x_1,\dots,x_n).eq(x_i,true),Solve(T')) & \\
compose_i(f_i(T,T'),0) \equiv 0 & \\
compose_i(f_i(T,T'),1) \equiv 1 & \\
compose_i(f_i(T,T'),T'') \equiv glue(T'',T,T') & \\
compose_i(f_j(T,T'),T'') \equiv f_j(compose_i(T,T''),compose_i(T',T'')) & 1 \leq i \neq j \leq n \\
glue(0,T,T') \equiv T & \\
glue(1,T,T') \equiv T' & \\
glue(f_i(T_1,T_2),T,T') \equiv f_i(glue(T_1,T,T'),glue(T_2,T,T')) &
\end{array}
$$

2.2 Regular graphs

A specification of regular graphs is obtained by turning REGTREE into a coalgebraic swinging type and the defined function *subs* of REGTREE into a destructor. The final model of the resulting specification contains *all* regular graphs constructed from $c_1,\dots,c_m$ and $f_1,\dots,f_n$:

REGGRAPH

`hidsorts`	$graph$	
`destructs`	$subs : graph \to \amalg_{i=1}^m 1 + \amalg_{i=1}^n graph^{k_i}$	subgraphs
	$sink : graph \to 1$	
`constructs`	$c_1,\dots,c_m :\to graph$	
	$f_1 : graph^{k_1} \to graph$	$k_1 > 0$
	$\dots$	
	$f_n : graph^{k_n} \to graph$	$k_n > 0$
`vars`	$T,T_1,\dots,T_{k_i} : graph$	$1 \leq i \leq n$
`Horn axioms`	$subs(c_i) \equiv \kappa_i()$	
	$subs(f_i(T_1,\dots,T_{k_i})) \equiv \kappa_{m+i}(T_1,\dots,T_{k_i})$	
	$sink(T) \equiv ()$	

The set of REGGRAPH-contexts is the smallest set CT of coterms such that $subs, sink \in CT$ and

$$
\begin{array}{rcl}
d : graph \to s \in CT & \Longrightarrow & \forall\ 1 \leq i \leq n,\ 1 \leq j \leq k_i : d \cdot \pi_j : graph^{k_i} \to s \in CT, \\
\{d_i : graph^{k_i} \to s_i\}_{i=1}^n \subseteq CT & \Longrightarrow & (\amalg_{i=1}^m id + \amalg_{i=1}^n d_i) \cdot subs : graph \to \amalg_{i=1}^m 1 + \amalg_{i=1}^n s_i \in CT.
\end{array}
$$

Let CSP be the cospecification of REGGRAPH and $C = Fin(visSP)$ (cf. [79], Def. 4.2.1). The *graph*-carrier of $Fin(CSP)$ is the set of all regular graphs with leaf labels $c_1, \dots, c_m$ and internal-node labels $f_1, \dots, f_n$: Let $P = \prod_{c:graph \to s \in CT} C_s$. $Fin(CSP)_{graph}$ is the greatest fixpoint of the function $\Phi : \wp(P) \to \wp(P)$ that is defined as follows: for all $A \in P$,

$$\Phi(A) \;=\; \{a \in A \mid \exists\, 1 \le i \le n,\; b \in A^{k_i}\; \forall\, d = (\amalg_{i=1}^m id + \amalg_{i=1}^n d_i) \cdot subs \in CT : \pi_d(a) = \kappa_{m+i}(\pi_{d_i}(b))\}.$$

The domain completion SP' of REGGRAPH contains the following additional axiom for each $a \in Fin(CSP)_{graph}$,

$$subs(a) \equiv \begin{cases} \kappa_i() & \text{if there is } 1 \le i \le n \text{ such that for all } d \in CT,\ \pi_d(a) = \kappa_i(), \\ \kappa_{m+i}(a_1, \dots, a_{k_i}) & \text{if there are } 1 \le i \le n \text{ and } a_1, \dots, a_{k_i} \text{ such that} \\ & \text{for all } d = (\amalg_{i=1}^m id + \amalg_{i=1}^n d_i) \cdot subs \in CT,\ \pi_d(a) = \kappa_{m+i}(\pi_{d_i}(a_1, \dots, a_{k_i})). \end{cases}$$

Since REGGRAPH has no assertions, REGGRAPH is cospec closed. [76], Korollar 6.1.5, [75], Thm. 5.15, and [79], Thm. 7.4, imply that REGGRAPH is functional, continuous and behaviorally consistent. Hence by [79], Thm. 4.2.5, $Fin(SP')$ and $Fin(CSP)$ are isomorphic.

Similarly to the step from REGTREE to REGGRAPH we turn the specification BDTREE (cf. Section 2.1) into a specification BDGRAPH of **binary decision diagrams** (BDDs):

BDGRAPH = BOOL `then`

`hidsorts`	bdd	
`destructs`	$subs : bdd \to 1 + 1 + \amalg_{i=1}^n (bdd \times bdd)$	
	$reduce : bdd \to bdd$	
	$sink : bdd \to 1$	
`constructs`	$0, 1 :\to bdd$	
	$f_1, \dots, f_n : bdd \times bdd \to bdd$	
	$\neg : bdd \to bdd$	
`vars`	$D, D', D_1, D_2, D_3, D_4 : bdd$	
`Horn axioms`	$subs(0) \equiv \kappa_1()$	
	$subs(1) \equiv \kappa_2()$	
	$subs(f_i(D, D')) \equiv \kappa_{2+i}(D, D')$	$1 \le i \le n$
	$subs(\neg D) \equiv \kappa_2() \;\Leftarrow\; subs(D) \equiv \kappa_1(z)$	
	$subs(\neg D) \equiv \kappa_1() \;\Leftarrow\; subs(D) \equiv \kappa_2(z)$	
	$subs(\neg D) \equiv \kappa_{2+i}(\neg D_1, \neg D_2) \;\Leftarrow\; subs(D) \equiv \kappa_{2+i}(D_1, D_2)$	
	$reduce(0) \equiv 0$	
	$reduce(1) \equiv 1$	
	$reduce(f_i(D, D')) \equiv f_i(reduce(D), reduce(D')) \;\Leftarrow\; D \not\equiv D'$	$1 \le i \le n$
	$reduce(f_i(D, D')) \equiv D \;\Leftarrow\; D \equiv D'$	$1 \le i \le n$
	$reduce(\neg D) \equiv \neg D$	
	$sink(D) \equiv ()$	
`assertions`	$subs(D) \equiv \kappa_1() \;\Rightarrow\; reduce(D) \equiv 0$	
	$subs(D) \equiv \kappa_2() \;\Rightarrow\; reduce(D) \equiv 1$	
	$(subs(D) \equiv \kappa_{2+i}(D_1, D_2) \wedge reduce(D_1) \equiv reduce(D_2))$	
	$\quad\Rightarrow\; reduce(D) \equiv reduce(D_1)$	
	$(subs(D) \equiv \kappa_{2+i}(D_1, D_2) \wedge reduce(D_1) \not\equiv reduce(D_2))$	
	$\quad\Rightarrow\; reduce(D) \equiv f_i(reduce(D_1), reduce(D_2))$	

In the domain completion of BDGRAPH, a BDD with k nodes comes as a collection of k constructor constants $d_1, \dots, d_k :\to bdd$ together with axioms of the form

$$subs(d_1) \equiv \kappa_{2+r_1}(d_{i_1}, d_{j_1}), \dots, subs(d_k) \equiv \kappa_{2+r_k}(d_{i_k}, d_{j_k})$$

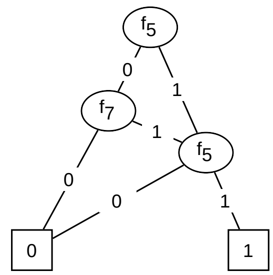

Figure 2. *A binary decision diagram.*

where $r_1,\dots,r_k \in \{1,\dots,n\}$ and $i_1,\dots,i_k,j_1,\dots,j_k \in \{0,1,d_1,\dots,d_k\}$. For instance, the BDD of Fig. 2 is represented by three constants d_1,d_2,d_3 and the axioms

$$subs(d_1) \equiv \kappa_{2+5}(d_2,d_3), \quad subs(d_2) \equiv \kappa_{2+7}(0,d_3), \quad subs(d_3) \equiv \kappa_{2+5}(0,1).$$

2.3 Graphs as functions

The nodes of a graph G that are reachable from a given node can be collected efficiently if G is represented as an *adjacency list*, i.e. a function that maps each node to the list of its direct successors. The following iterative programs for depthfirst search and checking acyclicity are adopted from [71], pp. 103 and 106.

GRAPH = LIST `then`

`hidsorts`	$list = list(entry)$
`hidsorts`	$graph = graph(entry) = entry \to list$
`destructs`	$apply : graph \times entry \to list$
`defuncts`	$depth : graph \times entry \to list$
	$depthLoop : graph \times list \times list \to list$
	$acyclicLoop : graph \times list^2 \times (1 + list) \to 1 + list$
`preds`	$acyclic : graph \times entry$
`vars`	$x, y : entry \;\; L, L', V, V'P : list \;\; V_1 : 1 + list \;\; G : graph$
`Horn axioms`	$depth(G,x) \equiv depthLoop(G,[x],[])$
	$depthLoop(G,[],V) \equiv V$
	$depthLoop(G,x:L,V) \equiv depthLoop(G,L,V) \;\; \Leftarrow \;\; x \in V$
	$depthLoop(G,x:L,V) \equiv depthLoop(G,G(x) +\!\!+ L,x:V) \;\; \Leftarrow \;\; x \notin V$
	$acyclic(G,x) \;\; \Leftarrow \;\; acyclicLoop(G,[x],[],([])) \equiv (V)$
	$acyclicLoop(G,[],P,V_1) \equiv V_1$
	$acyclicLoop(G,x:L,P,(V)) \equiv () \;\; \Leftarrow \;\; x \in P$
	$acyclicLoop(G,x:L,P,(V)) \equiv acyclicLoop(G,L,P,(V)) \;\; \Leftarrow \;\; x \notin P \wedge x \in V$
	$acyclicLoop(G,x:L,P,(V)) \equiv acyclicLoop(G,L,P,([x]))$
	$\quad \Leftarrow \;\; x \notin P \wedge x \notin V \wedge G(x) \equiv []$
	$acyclicLoop(G,x:L,P,(V)) \equiv acyclicLoop(G,L,P,(x:V'))$
	$\quad \Leftarrow \;\; x \notin P \wedge x \notin V \wedge G(x) \equiv y:L' \wedge acyclicLoop(G,y:L',x:P,(V)) \equiv (V')$
	$acyclicLoop(G,x:L,P,(V)) \equiv acyclicLoop(G,L,P,())$
	$\quad \Leftarrow \;\; x \notin P \wedge x \notin V \wedge G(x) \equiv y:L' \wedge acyclicLoop(G,y:L',x:P,(V)) \equiv ()$
	$acyclicLoop(G,x:L,P,()) \equiv ()$

2.4 Parsers

Maybe the most general definition of a parser or recognizer goes as follows. Given a signature $\Sigma = (S, F, \emptyset)$ and a Σ-structure A, a **parser for** A is an S-sorted function $parse : A \to \wp(T_\Sigma)$ such that for all $a \in A$ and $t \in parse(a)$, $t^A = a$.[2] For instance, parsers for a context-free language $G = (N, T, P)$ fit into this schema if one defines Σ as the abstract syntax for G and A as the following Σ-structure:

- For all $s \in S$, A_s is the set of $w \in T^*$ such that $s \xrightarrow{+}_G w$.
- For all $n \in \mathbb{N}$, $s, s_1, \dots, s_n \in N$, $w, w_1, \dots, w_n \in T^*$, $p = (s \longrightarrow ws_1w_1 \dots s_nw_n) \in P$, $1 \le i \le n$ and $a_i \in A_{s_i}$, $p^A(a_1, \dots, a_n) = wa_1w_1 \dots a_nw_n$.

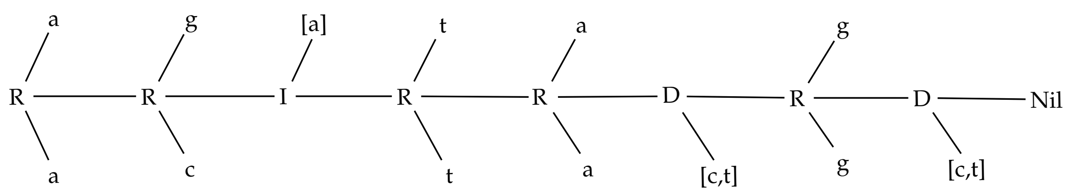

Figure 3. *A WFA-normal form.*

DNA sequence alignment problems [96, 26] can be solved by parsers for *pairs* of words over A. Here Σ is given by constructors for the grammar of *well-formed alignments* presented in [26] (p. 13). The interpretation of Σ-terms in A is defined in terms of axioms for the defined function *yield* (cf. [26], p. 10):

WFA = BASE(*base*) `and` LIST `then`

`hidsorts`	$match \quad insert \quad delete \quad align = match + insert + delete \quad noIns = match + delete$
	$noDel = match + insert$
`constructs`	$I : noIns \times list(base) \to insert$
	$D : list(base) \times noDel \to delete$
	$E : \to match$
	$R : base \times align \times base \to match$
`defuncts`	$yield : align \to list(base) \times list(base)$
	$yield : match \to list(base) \times list(base)$
	$yield : insert \to list(base) \times list(base)$
	$yield : delete \to list(base) \times list(base)$
`vars`	$x, y : base \quad a : align \quad m : match \quad d : delete \quad i : insert \quad L, L_1, L_2 : list(base)$
`Horn axioms`	$yield((m)) \equiv yield(m)$
	$yield((i)) \equiv yield(i)$
	$yield((d)) \equiv yield(d)$
	$yield(E) \equiv ([], [])$
	$yield(R(x, a, y)) \equiv (x : L_1, y : L_2) \quad \Leftarrow \quad yield(a) \equiv (L_1, L_2)$
	$yield(I((m), L)) \equiv (L_1, L +\!\!+ L_2) \quad \Leftarrow \quad yield(m) \equiv (L_1, L_2)$
	$yield(I((d), L)) \equiv (L_1, L +\!\!+ L_2) \quad \Leftarrow \quad yield(d) \equiv (L_1, L_2)$
	$yield(D(L, (m))) \equiv (L +\!\!+ L_1, L_2) \quad \Leftarrow \quad yield(m) \equiv (L_1, L_2)$
	$yield(D(L, (i))) \equiv (L +\!\!+ L_1, L_2) \quad \Leftarrow \quad yield(i) \equiv (L_1, L_2)$

Given a pair (L_1, L_2) of base sequences, a parser for *yield* enumerates all WFA-normal forms t such that $yield(t) \equiv_{WFA} (L_1, L_2)$. The parser developed in [26], Section 2.4, follows the schema of dynamic programming, i.e. tabulates recursive calls. We develop the parser in three steps. The first one (WFA-Parser0) works directly

[2] For all S-sorted sets A, let $\wp(A)_s = \wp(A_s)$ for all $s \in S$.

on the pair of sequences to be aligned (and their suffixes passed to recursive calls). The second one submits the positions of the suffixes with respect to the entire sequences instead of the suffixes themselves. The third one replaces the definition of the parser as a set of recursive functions (on pairs of numbers) by an equivalent definition of recursively defined tables. The specification BASE(*base*) is supposed to include axioms for the actual alignment relation $\sim: base^2$ between elements of the first and the second sequence, respectively.

Let us first collect auxiliary list operations to be used in the parser(s) in an extension of LIST (cf. Section 1.2):

LIST2 = LIST `then`

`hidsorts` $list = list(entry)$

`defuncts` $parts2 : list \to list(list^2)$
$parts3 : list \to list(list^3)$
$[\_..\_] : nat \times nat \to list(nat)$
$mkPairs : nat \times nat \to list(nat^2)$

`vars` $x : entry \;\; L, L_1, L_2 : list \;\; i, j : nat$

`Horn axioms`
$parts2([]) \equiv []$
$parts2(x : L) \equiv ([x], L) : map(\lambda(L_1, L_2).(x : L_1, L_2))(parts2(L))$
$parts3([]) \equiv []$
$parts3([x]) \equiv [([], [x], [])]$
$parts3(x : y : L) \equiv map(\lambda(L_1, L_2).([], x : L_1, L_2))(parts2(y : L)) \;++$
$\quad map(\lambda(L_1, L_2).([x], L_1, L_2))(parts2(y : L)) \;++$
$\quad map(\lambda(L, L_1, L_2).(x : L, L_1, L_2))(parts3(y : L))$
$[i..j] \equiv i : [i + 1..j] \quad \Leftarrow \quad i < j$
$[i..j] \equiv [] \quad \Leftarrow \quad i \geq j$
$mkPairs(i, j) \equiv (i, j) : (mkPairs(i + 1, j)`join`mkPairs(i, j - 1)) \quad \Leftarrow \quad i < j$
$mkPairs(i, j) \equiv [] \quad \Leftarrow \quad i \geq j$

WFA-Parser0 = LIST2 `and` WFA `then`

`defuncts` $align : list(base)^2 \to list(align)$
$match : list(base)^2 \to list(match)$
$insert : list(base)^2 \to list(insert)$
$delete : list(base)^2 \to list(delete)$
$mkIns : list(base) \to noIns \to insert$
$mkDel : list(base) \to noDel \to delete$

`vars` $x, y : base \;\; L, L' : list(base) \;\; a : align \;\; t : noIns \;\; u : noDel$

`Horn axioms`
$align(L, L') \equiv map((-))(match(L, L') ++ insert(L, L') ++ delete(L, L'))$
$match([], []) \equiv [E]$
$match([], x : L) \equiv []$
$match(x : L, []) \equiv []$
$match(x : L, y : L') \equiv map(\lambda a.R(x, a, y))(align(L, L')) \quad \Leftarrow \quad x \sim y$
$match(x : L, y : L') \equiv [] \quad \Leftarrow \quad x \not\sim y$
$insert(L, L') \equiv concatMap(g)(parts2(L'))$
$\quad \Leftarrow \quad \forall\, L_1, L_2 : g(L_1, L_2) \equiv map(mkIns(L_1))(match(L, L_2) ++ delete(L, L_2))$
$delete(L, L') \equiv concatMap(g)(parts2(L))$
$\quad \Leftarrow \quad \forall\, L_1, L_2 : g(L_1, L_2) \equiv map(mkDel(L_1))(match(L_2, L') ++ insert(L_2, L'))$
$mkIns(L)(t) \equiv I(t, L)$
$mkDel(L)(u) \equiv D(L, u)$

A dynamic-programming version similar to the one presented in [26] can be derived systematically from WFA-Parser0. At first, the induction on the decreasing suffixes of L and L' is replaced by an induction on the initial positions i and k of the suffixes within L resp. L'. Since this parser does not modify them, the sequences L or L' are declared as constants and thus need not be forwarded to the parse functions as parameters.

WFA-Parser1 = LIST2 `and` WFA `then`

`defuncts` $L, L' :\to list(base)$
$j, l :\to nat$
$parse :\to list(align)$
$align : nat^2 \to list(align)$
$match : nat^2 \to list(match)$
$insert : nat^2 \to list(insert)$
$delete : nat^2 \to list(delete)$
$mkIns : list(base) \to noIns \to insert$
$mkDel : list(base) \to noDel \to delete$

`vars` $x, y : base \;\; i, k, m : nat \;\; a : align \;\; t : noIns \;\; u : noDel$

`Horn axioms` $j \equiv length(L)$
$l \equiv length(L')$
$parse \equiv align(0, 0)$
$align(i, k) \equiv map((-))(match(i, k) +\!\!+ insert(i, k) +\!\!+ delete(i, k))$
$match(i, k) \equiv [E] \quad \Leftarrow \quad i \geq j \wedge k \geq l$
$match(i, k) \equiv [] \quad \Leftarrow \quad i \geq j \wedge k < l$
$match(i, k) \equiv [] \quad \Leftarrow \quad i < j \wedge k \geq l$
$match(i, k) \equiv map(\lambda a.R(L!!i, a, L'!!k))(align(i + 1, k + 1))$
$\quad \Leftarrow \quad i < j \wedge k < l \wedge L!!i \sim L'!!k$
$match(i, k) \equiv [] \quad \Leftarrow \quad i < j \wedge k < l \wedge L!!i \not\sim L'!!k$
$insert(i, k) \equiv concatMap(g)([k + 1..l])$
$\quad \Leftarrow \;\; \forall\, m : g(m) \equiv map(mkIns(sublist(L')(k, m)))(match(i, m) +\!\!+ delete(i, m))$
$delete(i, k) \equiv concatMap(g)([i + 1..j])$
$\quad \Leftarrow \;\; \forall\, m : g(m) \equiv map(mkDel(sublist(L)(i, m)))(match(m, k) +\!\!+ insert(m, k))$
$mkIns(L)(t) \equiv I(t, L)$
$mkDel(L)(u) \equiv D(L, u)$

In the second development step, tables implement the parse functions *align*, *match*, *insert* and *delete* (cf. Section 1.2.4). Recursive calls of the parse functions are replaced by corresponding table lookups.

WFA-Parser2 = LIST2 `and` WFA `then`

`hidsorts` $table = table(list(base))$

`constructs` $mkTab : (nat^2 \to list(base)) \to table$
$align :\to table(list(align))$
$match :\to table(list(match))$
$insert :\to table(list(insert))$
$delete :\to table(list(delete))$

`destructs` $\_!\_ : table \times nat^2 \to list(base)$

`defuncts` $L, L' :\to list(base)$
$j, l :\to nat$
$parse :\to list(align)$
$mkIns : list(base) \to noIns \to insert$

$mkDel : list(base) \to noDel \to delete$

`vars` $i, j, k, l, m, n : nat \quad f : nat^2 \to list(base) \quad x, y : base \quad a : align \quad t : noIns \quad u : noDel$

`Horn axioms`

$$mkTab(f)!(m,n) \equiv f(m,n) \quad \Leftarrow \quad 0 \le m \le j \wedge 0 \le n \le l$$
$$mkTab(f)!(m,n) \equiv [] \quad \Leftarrow \quad 0 > m \vee m > j \vee 0 > n \vee n > l$$
$$j \equiv length(L)$$
$$l \equiv length(L')$$
$$parse \equiv align!(0,0)$$
$$align \equiv mkTab(f) \Leftarrow \forall\, i,k : f(i,k) \equiv \; map((-))(match!(i,k) +\!\!+ insert!(i,k) +\!\!+ delete!(i,k))$$
$$\begin{aligned} match \equiv mkTab(f) \Leftarrow\ & \forall\, i,k : i \ge j \wedge k \ge l \Rightarrow f(i,k) \equiv [E] \wedge \\ & \forall\, i,k : i \ge j \wedge k < l \Rightarrow f(i,k) \equiv [] \wedge \\ & \forall\, i,k : i < j \wedge k \ge l \Rightarrow f(i,k) \equiv [] \wedge \\ & \forall\, i,k : i < j \wedge k < l \wedge L!!i \sim L'!!k \\ & \qquad \Rightarrow f(i,k) \equiv map(\lambda a.R(L!!i, a, L'!!k))(align!(i+1,k+1)) \wedge \\ & \forall\, i,k : i < j \wedge k < l \wedge L!!i \not\sim L'!!k \Rightarrow f(i,k) \equiv [] \end{aligned}$$
$$\begin{aligned} insert \equiv mkTab(f) \Leftarrow\ & \forall\, i,k : concatMap(g)([k+1..l]) \wedge \\ & \forall\, m : g(m) \equiv map(mkIns(sublist(L')(k,m)))(match!(i,m) +\!\!+ delete!(i,m)) \end{aligned}$$
$$\begin{aligned} delete \equiv mkTab(f) \Leftarrow\ & \forall\, i,k : concatMap(g)([i+1..j]) \wedge \\ & \forall\, m : g(m) \equiv map(mkDel(sublist(L)(i,m)))(match!(m,k) +\!\!+ insert!(m,k)) \end{aligned}$$
$$mkIns(L)(t) \equiv I(t, L)$$
$$mkDel(L)(u) \equiv D(L, u)$$

In a similar way, the following *yield* function provides the basis for recognizing **local separated palindromes** [27, 28]. While the *yield* function of WFA returns *pairs* of words, the *yield* function of LSP delivers single words:

LSP = BASE(*base*) `and` LIST `then`

`hidsorts` $match \quad context \quad start$

`constructs`

$$E : list(base) \to start$$
$$S : context \to start$$
$$C : list(base) \times match \times list(base) \to context$$
$$P : base \times align \times base \to match$$
$$PI : base \times align \times base \to align$$
$$I : list(base) \to align$$

`defuncts`

$$yield : start \to list(base)$$
$$yield : context \to list(base)$$
$$yield : match \to list(base)$$
$$yield : align \to list(base)$$

`vars` $x, y : base \quad m : match \quad i : align \quad c : context \quad L, L_1, L_2 : list(base)$

`Horn axioms`

$$yield(E(L)) \equiv L$$
$$yield(S(c)) \equiv yield(c)$$
$$yield(C(L_1, m, L_2)) \equiv L_1 +\!\!+ yield(m) +\!\!+ L_2$$
$$yield(P(x, m, y)) \equiv x : yield(m) +\!\!+ [y]$$
$$yield(PI(x, i, y)) \equiv x : yield(i) +\!\!+ [y]$$
$$yield(I(L)) \equiv L$$

BASE(*base*) is supposed to include axioms for the alignment relation $\sim : base^2$ between individual elements of the sequence to be parsed. For recognizing palindromes, $\sim$ must be defined as equality.

LSP-Parser0 = LIST2 `and` LSP `then`

`defuncts`
$start : list(base) \to list(start)$
$context : list(base) \to list(context)$
$match : list(base) \to list(match)$
$align : list(base) \to list(align)$
$mkCon : list(base)^2 \to match \to context$

`vars`
$x, y : base \;\; L, L' : list(base) \;\; t : match$

`Horn axioms`
$start([]) \equiv [E([])]$
$start([x]) \equiv [E([x])]$
$start(x : y : L) \equiv E(x : y : L) : map(S)(context(x : y : L))$
$context(L) \equiv concatMap(g)(parts3(L))$
$\quad \Leftarrow \;\; \forall\, L_1, L, L_2 : g(L_1, L, L_2) \equiv map(mkCon(L_1, L_2))(match(L))$
$match([]) \equiv []$
$match([x]) \equiv []$
$match(x : y : L) \equiv map(\lambda i.P(x, i, last(y : L)))(align(init(y : L))) \quad \Leftarrow \quad x \sim last(y : L)$
$match(x : y : L) \equiv [] \quad \Leftarrow \quad x \not\sim last(y : L)$
$align([]) \equiv [I([])]$
$align([x]) \equiv [I([x])]$
$align(x : y : L) \equiv map(\lambda m.PI(x, i, last(y : L)))(align(init(y : L))) \quad \Leftarrow \quad x \sim last(y : L)$
$align(x : y : L) \equiv [I(x : y : L)] \quad \Leftarrow \quad x \not\sim last(y : L)$
$mkCon(L_1, L_2)(t) \equiv C(L_1, t, L_2)$

LSP-Parser1 = LIST2 `and` LSP `then`

`defuncts`
$L : \to list(base)$
$n : \to nat$
$parse : \to list(start)$
$start : nat^2 \to list(start)$
$context : nat^2 \to list(context)$
$match : nat^2 \to list(match)$
$align : nat^2 \to list(align)$
$mkNoPal : nat^2 \to start$
$mkInner : nat^2 \to align$
$mkCon : list(base)^2 \to match \to context$

`vars`
$x, y : base \;\; i, j, k, l : nat \;\; t : match \;\; u : align$

`Horn axioms`
$n \equiv length(L)$
$parse \equiv start(0, n)$
$start(i, j) \equiv [mkNoPal(i, j)] \quad \Leftarrow \quad i \geq j$
$start(i, j) \equiv mkNoPal(i, j) : map(S)(context(i, j)) \quad \Leftarrow \quad i < j$
$context(i, j) \equiv concatMap(g)(mkPairs(i, j))$
$\quad \Leftarrow \;\; \forall\, k, l : g(k, l) \equiv map(mkCon(i, k, l, j))(match(k, l))$
$match(i, j) \equiv [] \quad \Leftarrow \quad i \geq j - 1$
$match(i, j) \equiv map(\lambda u.P(L!!i, u, L!!j))(align(i + 1, j - 1)) \quad \Leftarrow \quad i < j - 1 \wedge L!!i \sim L!!(j - 1)$
$match(i, j) \equiv [] \quad \Leftarrow \quad i < j - 1 \wedge L!!i \not\sim L!!(j - 1)$
$align(i, j) \equiv [mkInner(i, j)] \quad \Leftarrow \quad i \geq j - 1$
$align(i, j) \equiv map(\lambda u.PI(L!!i, u, L!!j))(align(i + 1, j - 1)) \quad \Leftarrow \quad i < j - 1 \wedge L!!i \sim L!!(j - 1)$

$$align(i,j) \equiv [mkInner(i,j)] \quad \Leftarrow \quad i < j-1 \wedge L!!i \not\sim L!!(j-1)$$
$$mkNoPal(i,j) \equiv E(sublist(L)(i,j))$$
$$mkInner(i,j) \equiv I(sublist(L)(i,j))$$
$$mkCon(i,k,l,j)(t) \equiv C(sublist(L)(i,k), sublist(L)(l,j))$$

LSP-Parser2 = LIST2 `and` LSP `then`

`hidsorts` $table = table(list(base))$

`constructs` $mkTab : (nat^2 \to list(base)) \to table$
$start :\to table(list(start))$
$context :\to table(list(context))$
$match :\to table(list(match))$
$align :\to table(list(align))$

`destructs` $\_!\_ : table \times nat^2 \to list(base)$

`defuncts` $L :\to list(base)$
$lg :\to nat$
$parse :\to list(start)$
$start : nat^2 \to list(start)$
$context : nat^2 \to list(context)$
$match : nat^2 \to list(match)$
$align : nat^2 \to list(align)$
$mkNoPal : nat^2 \to start$
$mkInner : nat^2 \to align$
$mkCon : list(base)^2 \to match \to context$

`vars` $i,j,k,l,m,n : nat \quad f : nat^2 \to list(base) \quad x,y : base \quad a : align \quad t : match \quad u : align$

`Horn axioms`
$$mkTab(f)!(m,n) \equiv f(m,n) \quad \Leftarrow \quad 0 \le m \le lg \wedge 0 \le n \le lg$$
$$mkTab(f)!(m,n) \equiv [] \quad \Leftarrow \quad 0 > m \vee m > lg \vee 0 > n \vee n > lg$$
$$lg \equiv length(L)$$
$$parse \equiv start!(0,lg)$$
$$start \equiv mkTab(f)$$
$$\Leftarrow \ \forall\, i,j : i \ge j \Rightarrow f(i,j) \equiv [mkNoPal(i,j)] \wedge$$
$$\forall\, i,j : i < j \Rightarrow f(i,j) \equiv mkNoPal(i,j) : map(S)(context!(i,j))$$
$$context \equiv mkTab(f)$$
$$\Leftarrow \ \forall\, i,j : f(i,j) \equiv concatMap(g)(mkPairs(i,j)) \wedge$$
$$\forall\, k,l : g(k,l) \equiv map(mkCon(i,k,l,j))(match!(k,l))$$
$$match \equiv mkTab(f)$$
$$\Leftarrow \ \forall\, i,j : i \ge j \Rightarrow f(i,j) \equiv [] \wedge$$
$$\forall\, i,j : i < j \wedge L!!i \sim L!!j \Rightarrow f(i,j) \equiv map(\lambda u.P(L!!i,u,L!!j))(align!(i+1,j-1))$$
$$\forall\, i,j : i < j \wedge L!!i \not\sim L!!j \Rightarrow f(i,j) \equiv []$$
$$align \equiv mkTab(f)$$
$$\Leftarrow \ \forall\, i,j : i \ge j \Rightarrow f(i,j) \equiv [mkInner(i,j)] \wedge$$
$$\forall\, i,j : i < j \wedge L!!i \sim L!!j \Rightarrow f(i,j) \equiv map(\lambda u.PI(L!!i,u,L!!j))(align!(i+1,j-1))$$
$$\forall\, i,j : i < j \wedge L!!i \not\sim L!!j \Rightarrow f(i,j) \equiv [mkInner(i,j)]$$
$$mkNoPal(i,j) \equiv E(sublist(L)(i,j))$$
$$mkInner(i,j) \equiv I(sublist(L)(i,j))$$
$$mkCon(i,k,l,j)(t) \equiv C(sublist(L)(i,k), sublist(L)(l,j))$$

Expander2 [80] provides Haskell implementations of WFA-Parser2 and LSP-Parser2 and a GUI for executing these algorithms and displaying their results.

3 State-based types

3.1 Bank accounts

We present swinging versions of a favorite example used for demonstrating formal approaches to the *object-oriented* or *state-based* specification of data types (cf., e.g., [33, 64]).

ACC_STATE = LIST **then**

`hidsorts`	$transaction \quad acc$
`objconstructs`	$from, to : entry \times nat \to transaction$
`constructs`	$new : entry \to acc$
	$credit, debit : acc \times nat \times entry \to acc$
`destructs`	$name : acc \to entry$
	$bal : acc \to nat$
	$record : acc \to list(transaction)$
`vars`	$x : entry \quad n : nat \quad a : acc$
`Horn axioms`	$name(new(x)) \equiv x$
	$bal(new(x)) \equiv 0$
	$record(new(x)) \equiv []$
	$name(credit(a, n, x)) \equiv name(a)$
	$bal(credit(a, n, x)) \equiv bal(a) + n$
	$record(credit(a, n, x)) \equiv from(x, n) : record(a)$
	$name(debit(a, n, x)) \equiv name(a)$
	$bal(debit(a, n, x)) \equiv bal(a) - n$
	$record(debit(a, n, x)) \equiv to(x, n) : record(a)$

Since behavioral *acc*-equivalence is determined by the destructors *name*, *bal* and *record*, the *acc*-carrier of the final ACC_STATE-model, say A, is isomorphic to the product $A_{entry} \times \mathbb{N} \times A^*_{transaction}$. Of course, this is also achieved by declaring *acc* as the product $entry \times nat \times list(transaction)$ and *name*, *bal* and *record* as the corresponding projections. To ensure that ACC_STATE is consistent, *new*, *credit* and *debit* had to be defined functions and would be axiomatized as follows:

$$\begin{aligned} new(x) &\equiv (x, 0, []) \\ credit(a, n, x) &\equiv (name(a), bal(a) + n, from(x, n) : record(a)) \\ debit(a, n, x) &\equiv (name(a), bal(a) - n, to(x, n) : record(a)) \end{aligned}$$

Unfortunately, these axioms could not be *inherited* to extensions of ACC_STATE that represent *subclasses* of *acc*-objects with additional *acc*-constructors. These would require the re-specification of all defined functions with *acc*-arguments. If, on the other hand, ACC_STATE is extended by *acc*-constructors, no axiom need not be removed. This "option for inheritance" is—besides the involvement of observers—the reason for calling ACC_STATE an object-oriented specification.

3.1.1 A functional version

ACC_LOCAL1 = ACC_STATE **then**

`hidsorts`	com
`objconstructs`	$deposit, withdraw : nat \to com$
	$send, receive : nat \times entry \to com$
`defuncts`	$\_ : \_ : acc \times com \to acc$

`vars`	$x : entry \quad n : nat \quad a : acc \quad c : com$
`Horn axioms`	$a : deposit(n) \equiv credit(a, n, name(a))$
	$a : withdraw(n) \equiv debit(a, n, name(a)) \quad \Leftarrow \quad n \le bal(a)$
	$a : withdraw(n) \equiv a \quad \Leftarrow \quad n > bal(a)$
	$a : send(n, x) \equiv debit(a, n, x) \quad \Leftarrow \quad n \le bal(a)$
	$a : send(n, x) \equiv debit(a, n, x) \quad \Leftarrow \quad n > bal(a)$
	$a : receive(n, x) \equiv credit(a, n, x)$

ACC_GLOBAL1 = ACC_LOCAL1 `and` SET `then`

`hidsorts`	$message$
`objconstructs`	$open, close : entry \to message$
	$\_.\_ : entry \times com \to message$
	$\_;\_ : message \times message \to message$
`defuncts`	$\_:\_ : set(acc) \times message \to set(acc)$
	$transfer : entry \times nat \times entry \to message$
`static preds`	$\_ \in \_ : entry \times set(acc)$
	$\_ \notin \_ : entry \times set(acc)$
ν`-preds`	$\_ \approx \_ : message \times message$
`vars`	$x, y : entry \quad n : nat \quad a : acc \quad as : set(acc) \quad c : com \quad m, m' : message$
`Horn axioms`	$as : open(x) \equiv as \cup \{new(x)\} \quad \Leftarrow \quad x \notin as$
	$as : open(x) \equiv as \quad \Leftarrow \quad x \in as$
	$as : close(x) \equiv as \setminus \{a\} \quad \Leftarrow \quad name(a) \equiv x$
	$as : x.c \equiv as \setminus \{a\} \cup \{a : c\} \quad \Leftarrow \quad a \in as \wedge name(a) \equiv x$
	$as : x.c \equiv as \quad \Leftarrow \quad x \notin as$
	$as : (m; m') \equiv (as : m) : m'$
	$transfer(x, n, y) \ \equiv \ x.send(n, y);\ y.receive(n, x)$
	$x \in as \quad \Leftarrow \quad name(a) \equiv x \wedge a \in as$
	$x \notin as \quad \Leftarrow \quad name(a) \equiv x \wedge a \notin as$
`co-Horn axioms`	$m \approx m' \quad \Rightarrow \quad as : m \sim as : m'$

Terms of the form $a : c$ or $as : m$ represent configurations that are typical for SOS (= *structural operational semantics*) rules (cf. [84]). An SOS rule is nothing but a Horn axiom for a dynamic predicate. Configurations are pairs consisting of a state (here: of single or several accounts) and a command (here: a *com*- resp. *message*-term).

3.1.2 A relational version with state sets

ACC_GLOBAL1 evaluates *acc*- and $set(acc)$-terms in a completely functional way. The following alternative specification ACC_GLOBAL2 has dynamic predicates $\longrightarrow$ and $\Longrightarrow$ instead of the function symbol(s) :. An equation of the form $a : c \equiv b$ becomes an atom $a \stackrel{c}{\longrightarrow} b$, which represents a transition between states of a single account. An equation of the form $as : m \equiv bs$ becomes an atom $as \stackrel{m}{\Longrightarrow} bs$, which represents a transition between states of several accounts. While the interpreter ":" of ACC_GLOBAL1 is a function and thus the command term c of $a : c \equiv b$ can only include *input* variables, the label c of $a \stackrel{c}{\longrightarrow} b$ may also contain *output* variables, which are instantiated during the transition. For instance, the evaluation of a command $bal?(n)$ shall produce a valuation of n. Moreover, since $\longrightarrow$ and $\Longrightarrow$ are predicates, we do not need counterparts of axioms for ":" whose only purpose was to totalize ":". For the same reason we may identify the label sorts *com* and *message*.

ACC_LOCAL2 = ACC_STATE `then`

`hidsorts`	com
`objconstructs`	$deposit, withdraw : nat \to com$
	$send, receive : nat \times entry \to com$
	$bal? : nat \to com$
`dynamic preds`	$\_ \overset{\_}{\longrightarrow} \_ : acc \times com \times acc$
`vars`	$x : entry \quad n : nat \quad a : acc \quad c : com$
`Horn axioms`	$a \overset{deposit(n)}{\longrightarrow} credit(a, n, name(a))$
	$a \overset{withdraw(n)}{\longrightarrow} debit(a, n, name(a)) \quad \Leftarrow \quad n \leq bal(a)$
	$a \overset{send(n,x)}{\longrightarrow} debit(a, n, x) \quad \Leftarrow \quad n \leq bal(a)$
	$a \overset{receive(n,x)}{\longrightarrow} credit(a, n, x)$
	$a \overset{bal?(n)}{\longrightarrow} a \quad \Leftarrow \quad n \equiv bal(a)$

ACC_GLOBAL2 = ACC_LOCAL2 `and` SET `then`

`objconstructs`	$open, close : entry \to com$
	$\_.\_ : entry \times com \to com$
	$\_;\_ : com \times com \to com$
`defuncts`	$transfer : entry \times nat \times entry \to com$
`dynamic preds`	$\_ \overset{\_}{\Longrightarrow} \_ : set(acc) \times com \times set(acc)$
`static preds`	$\_ \notin \_ : entry \times set(acc)$
ν`-preds`	$\_ \approx \_ : com \times com$
`vars`	$x, y : entry \quad a, a' : acc \quad as, bs, as', bs' : set(acc) \quad c, c' : com$
`Horn axioms`	$as \overset{open(x)}{\Longrightarrow} as \cup \{new(x)\} \quad \Leftarrow \quad x \notin as$
	$as \overset{close(x)}{\Longrightarrow} as \setminus \{a\} \quad \Leftarrow \quad name(a) \equiv x$
	$as \overset{x.c}{\Longrightarrow} as \setminus \{a\} \cup \{a'\} \quad \Leftarrow \quad a \overset{c}{\longrightarrow} a' \wedge a \in as \wedge name(a) \equiv x$
	$as \overset{c;c'}{\Longrightarrow} as' \quad \Leftarrow \quad as \overset{c}{\Longrightarrow} bs \wedge bs \overset{c'}{\Longrightarrow} as'$
	$transfer(x, n, y) \ \equiv \ x.send(n, y);\ y.receive(n, x)$
	$x \notin as \quad \Leftarrow \quad name(a) \equiv x \wedge a \notin as$
`co-Horn axioms`	$c \approx c' \quad \Rightarrow \quad (as \overset{c}{\Longrightarrow} bs \ \Rightarrow \ \exists\ bs' : (as \overset{c'}{\Longrightarrow} bs' \wedge bs \sim bs'))$
	$c \approx c' \quad \Rightarrow \quad (as \overset{c'}{\Longrightarrow} bs' \ \Rightarrow \ \exists\ bs : (as \overset{c}{\Longrightarrow} bs \wedge bs \sim bs'))$

Command sequences such as

$$open(x);\ x.deposit(100);\ x.withdraw(50);\ x.bal?(n) \tag{1}$$

represent imperative programs. For instance, (1) is executed by unfolding atom (2):

$$as \overset{open(x);\ x.deposit(100);\ x.withdraw(50);\ x.bal?(n)}{\Longrightarrow} bs \tag{2}$$

predicate unfolding

$\vdash \ \exists\ as_1, as_2, as_3 : as \overset{open(x)}{\Longrightarrow} as_1 \wedge as_1 \overset{x.deposit(100)}{\Longrightarrow} as_2 \wedge as_2 \overset{x.withdraw(50)}{\Longrightarrow} as_3 \wedge as_3 \overset{x.bal?(n)}{\Longrightarrow} bs$

predicate unfolding and expansion with $as \sim as$

$\vdash \ \exists\ as_2, as_3 : x \notin as \wedge as \cup \{new(x)\} \overset{x.deposit(100)}{\Longrightarrow} as_2 as_2 \overset{x.withdraw(50)}{\Longrightarrow} as_3 \wedge as_3 \overset{x.bal?(n)}{\Longrightarrow} bs$

predicate unfolding

$\vdash \ \exists\ a_1, a_2, as_3 : x \notin as \wedge a_1 \equiv new(x) \wedge a_1 \overset{deposit(100)}{\longrightarrow} a_2 \wedge as \cup \{a_1\} \setminus \{a_1\} \cup \{a_2\} \overset{x.withdraw(50)}{\Longrightarrow} as_3 \wedge as_3 \overset{x.bal?(n)}{\Longrightarrow} bs$

behavioral term replacement

$\vdash \ \exists\ a_2, as_3 : x \notin as \wedge new(x) \overset{deposit(100)}{\longrightarrow} a_2 \wedge as \cup \{a_2\} \overset{x.withdraw(50)}{\Longrightarrow} as_3 \wedge as_3 \overset{x.bal?(n)}{\Longrightarrow} bs$

predicate and function unfolding

$\vdash\ \exists\, as_3 : x \notin as \wedge as \cup \{credit(new(x), 100, x)\} \stackrel{x.withdraw(50)}{\Longrightarrow} as_3 \wedge as_3 \stackrel{x.bal?(n)}{\Longrightarrow} bs$

predicate unfolding

$\vdash\ \exists\, a_2, a_3 : x \notin as \wedge a_2 \equiv credit(new(x), 100, x) \wedge a_2 \stackrel{withdraw(50)}{\longrightarrow} a_3 \wedge as \cup \{a_2\} \setminus \{a_2\} \cup \{a_3\} \stackrel{x.bal?(n)}{\Longrightarrow} bs$

behavioral term replacement

$\vdash\ \exists\, a_3 : x \notin as \wedge credit(new(x), 100, x) \stackrel{withdraw(50)}{\longrightarrow} a_3 \wedge as \cup \{a_3\} \stackrel{x.bal?(n)}{\Longrightarrow} bs$

predicate and function unfolding

$\vdash\ x \notin as \wedge as \cup \{debit(credit(new(x), 100, x), 50, x)\} \stackrel{x.bal?(n)}{\Longrightarrow} bs$

predicate and function unfolding

$\vdash\ \exists\, a_3, b : x \notin as \wedge a_3 \equiv debit(credit(new(x), 100, x), 50, x) \wedge a_3 \stackrel{bal?(n)}{\longrightarrow} b \wedge as \cup \{a_3\} \setminus \{a_3\} \cup \{b\} \equiv bs$

predicate unfolding

$\vdash\ x \notin as \wedge n \equiv bal(debit(credit(new(x), 100, x), 50, x)) \wedge \dots$

function unfolding

$\vdash\ x \notin as \wedge n \equiv 50 \wedge \dots$

Hence the derivation computes a solution of (2) in n.

3.1.3 A relational version with ports

A third version of the specification follows the paradigm of concurrent logic programming introduced by [97], namely to implement objects as command stream consuming predicates. This was also adopted by [38] for the functional-logic language Curry where it has recently been combined with message passing via ports in the sense of [48, 39]. Following [48] we specify ports as collections (here: lists) of commands in a way that abstracts from the particular way several command streams are merged into a single input stream.

An object predicate *Ob* has two arguments: a list *acts* of actions, commands, etc., to be processed and an object state s. Intuitively, the **object atom** $Ob(acts, s)$ is true if *acts* leads from s to a final state.[3] The object-as-predicate paradigm complies with the state-as-hidden-term concept employed in, e.g., ACC_GLOBAL1 and ACC_GLOBAL2. ACC_GLOBAL3 adds object predicates A and AS that represent *acc*- resp. *accs*-objects. An object is created whenever a Horn axiom $r(t) \Leftarrow \varphi$ is applied and φ contains an object predicate $\neq r$ (see, e.g., axiom $(*)$ below). ACC_GLOBAL3 interprets *accs* not as account sets, but as maps assigning ports to account names:

ACC_GLOBAL3 = ACC_LOCAL2 `and` LIST `and` MAP `then`

`hidsorts`	$accs = map(entry, list(com))$
`objconstructs`	$open, close : entry \to com$
	$\_.\_ : entry \times com \to com$
`defuncts`	$transfer : entry \times nat \times entry \to com$
`static preds`	$A : list(com) \times acc$
	$AS : list(com) \times accs$
	$free : list(com)$
`dynamic preds`	$\_ \stackrel{\_}{\Longrightarrow} \_ : set(acc) \times com \times set(acc)$
ν`-preds`	$\_ \approx \_ : list(com) \times list(com)$
`vars`	$x, y : entry \quad c, c' : com \quad cL, cL' : list(com) \quad a, a' : acc \quad as, bs, as' : accs$
`Horn axioms`	$A([], a)$
	$A(c : cL, a) \quad \Leftarrow \quad a \stackrel{c}{\longrightarrow} a' \wedge A(cL, a')$
	$AS([], as)$
	$AS(c : cL, as) \quad \Leftarrow \quad as \stackrel{c}{\Longrightarrow} as' \wedge AS(cL, as')$

[3] In ACC_GLOBAL3, each state (= *acc*- or *accs*-term) is final.

$$
\begin{array}{lll}
& free(cL) \Leftarrow \text{"create a reference } cL\text{"} & \\
& as \overset{open(x)}{\Longrightarrow} upd(x,cL,as) \quad \Leftarrow \quad get(as,x) \equiv () \wedge free(cL) \wedge A(cL,new(x)) & (*) \\
& \qquad \text{A port } cL \text{ is created and assigned to } x\text{. The commands} & \\
& \qquad \text{arriving at } cL \text{ are processed by } A \text{ in state } new(x)\text{.} & \\
& as \overset{close(x)}{\Longrightarrow} remove(x,as) & \\
& as \overset{x.c}{\Longrightarrow} as \quad \Leftarrow \quad get(as,x) \equiv (cL) \wedge c \in cL & \\
& \qquad \text{The command } c \text{ is sent to } cL\text{, the port of } x\text{.} & \\
& transfer(x,n,y) \ \equiv \ x.send(n,y) : y.receive(n,x) : [] & \\
\texttt{co-Horn axioms} & cL \approx cL' \ \Rightarrow \ (AS(cL,as) \ \Rightarrow \ \exists\, as' : (AS(cL',as') \wedge as \sim as')) & \\
& cL \approx cL' \ \Rightarrow \ (AS(cL',as') \ \Rightarrow \ \exists\, as : (AS(cL,as) \wedge as \sim as')) &
\end{array}
$$

The command sequence (1) of Section 3.1.2 is executed in terms of ACC_GLOBAL3 as follows.

$AS(open(x) : x.deposit(100) : x.withdraw(50) : x.bal?(n) : [], as)$

predicate unfolding

$\vdash \ \exists\, cL : get(as,x) \equiv () \wedge A(cL,new(x)) \wedge$
$\quad AS(x.deposit(100) : x.withdraw(50) : x.bal?(n) : [], upd(x,cL,as))$

predicate unfolding

$\vdash \ \exists\, cL, cL' : get(as,x) \equiv () \wedge A(cL,new(x)) \wedge get(upd(x,b,as),x) \equiv (cL') \wedge$
$\quad deposit(100) \in cL' \wedge AS(x.withdraw(50) : x.bal?(n) : [], upd(x,cL,as))$

function unfolding

$\vdash \ \exists\, cL, cL' : get(as,x) \equiv () \wedge A(cL,new(x)) \wedge (cL) \equiv (cL') \wedge$
$\quad deposit(100) \in cL' \wedge AS(x.withdraw(50) : x.bal?(n) : [], upd(x,cL,as))$

constructor elimination

$\vdash \ \exists\, cL, cL' : get(as,x) \equiv () \wedge A(cL,new(x)) \wedge cL \equiv cL' \wedge$
$\quad deposit(100) \in cL' \wedge AS(x.withdraw(50) : x.bal?(n) : [], upd(x,cL,as))$

variable elimination

$\vdash \ \exists\, cL : get(as,x) \equiv () \wedge A(cL,new(x)) \wedge$
$\quad deposit(100) \in cL \wedge AS(x.withdraw(50) : x.bal?(n) : [], upd(x,cL,as))$

expansion with $c \in c : cL$

$\vdash \ \exists\, cL : get(as,x) \equiv () \wedge A(deposit(100) : cL, new(x)) \wedge$
$\quad AS(x.withdraw(50) : x.bal?(n) : [], upd(x,deposit(100) : cL,as))$

predicate and function unfolding

$\vdash \ \exists\, cL : get(as,x) \equiv () \wedge A(cL, credit(new(x),100,x)) \wedge$
$\quad AS(x.withdraw(50) : x.bal?(n) : [], upd(x,deposit(100) : cL,as))$

predicate unfolding

$\vdash \ \exists\, cL, cL' : get(as,x) \equiv () \wedge A(cL, credit(new(x),100,x)) \wedge$
$\quad get(upd(x,deposit(100) : cL,as),x) \equiv (cL') \wedge$
$\quad withdraw(50) \in cL' \wedge AS(x.bal?(n) : [], upd(x,deposit(100) : cL,as))$

function unfolding

$\vdash \ \exists\, cL, cL' : get(as,x) \equiv () \wedge A(cL, credit(new(x),100,x)) \wedge (deposit(100) : cL) \equiv (cL') \wedge$
$\quad withdraw(50) \in cL' \wedge AS(x.bal?(n) : [], upd(x,deposit(100) : cL,as))$

constructor elimination

$\vdash \ \exists\, cL, cL' : get(as,x) \equiv () \wedge A(cL, credit(new(x),100,x)) \wedge deposit(100) : cL \equiv cL' \wedge$
$\quad withdraw(50) \in cL' \wedge AS(x.bal?(n) : [], upd(x,deposit(100) : cL,as))$

variable elimination

$\vdash \ \exists\, cL : get(as,x) \equiv () \wedge A(cL, credit(new(x),100,x)) \wedge$
$\quad withdraw(50) \in deposit(100) : cL \wedge AS(x.bal?(n) : [], upd(x,deposit(100) : cL,as))$

expansion with $c \in c' : c : cL$

$$\vdash \exists\, cL : get(as,x) \equiv () \wedge A(withdraw(50) : cL, credit(new(x), 100, x)) \wedge AS(x.bal?(n) : [], upd(x, deposit(100) : withdraw(50) : cL, as))$$

predicate and function unfolding

$$\vdash \exists\, cL : get(as,x) \equiv () \wedge A(cL, debit(credit(new(x), 100, x), 50, x)) \wedge AS(x.bal?(n) : [], upd(x, deposit(100) : withdraw(50) : cL, as))$$

predicate unfolding

$$\vdash \exists\, cL, cL' : get(as,x) \equiv () \wedge A(cL, debit(credit(new(x), 100, x), 50, x)) \wedge get(upd(x, deposit(100) : withdraw(50) : cL, as), x) \equiv (cL') \wedge bal?(n) \in cL' \wedge AS([], upd(x, deposit(100) : withdraw(50) : cL, as))$$

predicate unfolding

$$\vdash \exists\, cL, cL' : get(as,x) \equiv () \wedge A(cL, debit(credit(new(x), 100, x), 50, x)) \wedge get(upd(x, deposit(100) : withdraw(50) : cL, as), x) \equiv (cL') \wedge bal?(n) \in cL'$$

function unfolding

$$\vdash \exists\, cL, cL' : get(as,x) \equiv () \wedge A(cL, debit(credit(new(x), 100, x), 50, x)) \wedge (deposit(100) : withdraw(50) : cL) \equiv (cL') \wedge bal?(n) \in cL'$$

constructor elimination

$$\vdash \exists\, cL, cL' : get(as,x) \equiv () \wedge A(cL, debit(credit(new(x), 100, x), 50, x)) \wedge deposit(100) : withdraw(50) : cL \equiv cL' \wedge bal?(n) \in cL'$$

variable elimination

$$\vdash \exists\, cL : get(as,x) \equiv () \wedge A(cL, debit(credit(new(x), 100, x), 50, x)) \wedge bal?(n) \in deposit(100) : withdraw(50) : cL$$

expansion with $c \in c' : c'' : c : cL$

$$\vdash \exists\, cL : get(as,x) \equiv () \wedge A(bal?(n) : cL, debit(credit(new(x), 100, x), 50, x))$$

predicate unfolding

$$\vdash \exists\, cL : get(as,x) \equiv () \wedge A(cL, debit(credit(new(x), 100, x), 50, x)) \wedge n \equiv 50$$

quantor elimination

$$\vdash get(as,x) \equiv () \wedge A([], debit(credit(new(x), 100, x), 50, x)) \wedge n \equiv 50$$

function and predicate unfolding

$$\vdash get(as,x) \equiv () \wedge n \equiv 50$$

3.2 Web scripting

The premise of axiom (a) of ACC_GLOBAL3 introduces the existentially quantified "free" variable cL that is supposed to be implemented as the creation of a reference. For axiom (b) to be "executed", cL must have been instantiated by a list that contains c. Such free variables also occur frequently in the specification of HTML documents [40] as expressions of the functional-logic language Curry [38]. Using ST syntax the expressions are built up as follows:

HTML = STRING `and` LIST `and` STATEMON `then`[4]

`hidsorts`	$Exp \quad Form = string \times list(Exp) \quad CgiRef = string$
	$Env = CgiRef \rightarrow String \quad Handler = Env \rightarrow M(Form)$
`objconstructs`	$Text : string \rightarrow Exp$
	$Struct : string \times list(string \times string) \times list(Exp) \rightarrow Exp$
	$Elem : string \times list(string \times string) \rightarrow Exp$
	$CRef : CgiRef \times Exp \rightarrow Exp$
	$Event : Exp \times handler \rightarrow Exp$
`defuncts`	$htext : string \rightarrow Exp$

$hrule : \to Exp$
$bold : list(Exp) \to Exp$
$textfield : CgiRef \times string \to Exp$
$button : string \times Handler \to Exp$
$revdup, guessform, nameform : \to M(Form)$
$guessinput : \to list(Exp)$
$revhandler, duphandler, guesshandler, firsthandler : CgiRef \to (Env \to M(Form))$
$lasthandler : CgiRef \times CgiRef \to (Env \to M(Form))$

`static preds` $free : CgiRef$

`vars` $r, r' : CgiRef \quad str : string \quad hexps : list(Exp) \quad handler : Handler \quad env : Env$

`Horn axioms` $htext(str) \equiv Text(str)$
$hrule \equiv Elem(\texttt{hr}, [])$
$bold(hexps) \equiv Struct(\texttt{b}, [], hexps)$
$textfield(r, str) \equiv CRef(r, Elem(\texttt{input}, [(\texttt{type}, \texttt{text}), (\texttt{name}, r), (\texttt{value}, str)]))$
$button(str, handler)$
$\quad \equiv \; Event(Elem(\texttt{input}, [(\texttt{type}, \texttt{submit}), (\texttt{name}, \texttt{event}), (\texttt{value}, str)]), handler)$
$revdup \equiv return(\texttt{Question}, [htext(\texttt{Enter a string}),$
$\qquad textfield(r, []),$
$\qquad hrule,$
$\qquad button(\texttt{Reverse string}, revhandler(r)),$
$\qquad button(\texttt{Duplicate string}, duphandler(r))])$
$\qquad\quad \Leftarrow \; free(r)$
$revhandler(r)(env) \equiv return(\texttt{Answer}, [htext(\texttt{Reversed input :} +\!\!+ rev(env(r)))])$
$duphandler(r)(env) \equiv return(\texttt{Answer}, [htext(\texttt{Duplicated input :} +\!\!+ env(r) +\!\!+ env(r))])$
$guessform \equiv return(\texttt{Number Guessing}, guessinput)$
$guessinput \equiv [htext(\texttt{Guess a number}),$
$\qquad textfield(r, []),$
$\qquad button(\texttt{Check}, guesshandler(r))] \quad \Leftarrow \quad free(r)$
$guesshandler(r)(env) \equiv return(\texttt{Answer}, [htext(\texttt{Bingo!})] +\!\!+ guessinput) \quad \Leftarrow \quad int(r) \equiv 42$
$guesshandler(r)(env) \equiv return(\texttt{Answer}, [htext(\texttt{Too small!}), hrule] +\!\!+ guessinput)$
$\quad \Leftarrow \; int(r) < 42$
$guesshandler(r)(env) \equiv return(\texttt{Answer}, [htext(\texttt{Too large!}), hrule] +\!\!+ guessinput)$
$\quad \Leftarrow \; int(r) > 42$
$nameform \equiv return(\texttt{First Name}, [htext(\texttt{Enter your first name}),$
$\qquad textfield(r, []),$
$\qquad button(\texttt{Continue}, firsthandler(r))])$
$\qquad\quad \Leftarrow \; free(r)$
$firsthandler(r)(env) \equiv return(\texttt{Last Name}, [htext(\texttt{Enter your first name}),$
$\qquad textfield(r', []),$
$\qquad button(\texttt{Continue}, lasthandler(r, r'))])$
$\qquad\quad \Leftarrow \; free(r')$
$lasthandler(r, r')(env) \equiv return(\texttt{Full Name}, [htext(env(r) +\!\!+ env(r'))])$
$free(r) \quad \Leftarrow \quad$ "create a reference r"

r and r' denote input elements of HTML forms.

[4] We write string constants in typewriter mode. $+\!\!+$ denotes string concatenation. $int(r)$ is the integer value of the string r. For STATEMON, cf. Section 1.2.5.

3.3 Plan formation

The three versions of ACC_GLOBAL (cf. Section 3.1) are coinductive (cf. [75]). Coinductive axioms yield a schema for SOS definitions as well as for algebraic nets, SDL systems (see Section 5), labelled transition logic (LTL; see [3]) and rewriting logic (*Maude* specifications; see [64]). These formalisms mainly differ with respect to the structure of a state. SOS and LTL regard states as (abstract) stores of values, while net states are tuples of multisets, usually called markings. SDL states are unstructured, but implicitly associated with valuations of program variables. States in Maude are sets of objects as in the above example (cf. [64], Section 12.4.1). Many formal approaches are two-tiered as they reason about transitions on the one hand and the structure of states on the other hand in different logical frameworks. However, LTL, swinging types and *modal fragments* of predicate logic (cf. [10]) adopt the one-tiered view of a labelled transition system as a ternary predicate of a data type specification.

Explicit state structures should be distinguished from the *agents* occurring in stream and process calculi. While the above specifications (and COM below) separate states from agents (= commands), process calculi identify them: an agent is both an abstract program and the initial state the program starts out from (see Section 4.4).

As in Section 3.1, object predicates are determined by transition systems:

$$\begin{aligned} Ob([], s) &\Leftarrow final(s), \\ Ob(act : acts, s) &\Leftarrow s \xrightarrow{act} s' \wedge Ob(acts, s'). \end{aligned}$$

$Ob(acts, s)$ holds true iff $acts$ leads from s to to a final state. If object predicates are used not for executing, but for generating action sequences, we deal with *plan formation*. This is accomplished by expanding implications of the form

$$initial(s) \quad \Rightarrow \quad Ob(acts, s).$$

The expansion has derived a plan if it results in a "solved" goal of the form $acts \equiv t$ where t is a normal form. This works because $acts$ is universally quantified in the formula $initial(s) \Rightarrow Ob(acts, s)$. A similar goal is to prove that final states are reachable from initial ones. This is achieved by expanding the formula

$$initial(s) \quad \Rightarrow \quad \exists\, acts : Ob(acts, s)$$

into *True*. Let us illustrate plan formation at three small examples.

Monkey wants banana. A monkey and a box are located at the door, at the window or in the middle of a room. In order to grasp the banana hanging from the ceiling in the middle of the room the monkey must go the box, push the box to the middle of the room and climb the box. The monkey-box system is in state (x, y, b) iff x is the position of the monkey, y is the position of the box and b is true iff the monkey is on the box.

MONKEY = LIST `then`

`sorts`	$position \quad state = position \times position \times bool \quad com$
`constructs`	$door, middle, window :\to position$
	$walk, push, climb :\to com$
`static preds`	$MB : list(com) \times state$
	$final : state$
	$\_ \xrightarrow{\_} \_ : state \times com \times state$
	$\not\equiv : position \times position$
`vars`	$x, y : position \quad act : com \quad acts : list(com) \quad s, s' : state$
`Horn axioms`	$MB([], s) \quad \Leftarrow \quad final(s)$
	$MB(act : acts, s) \quad \Leftarrow \quad s \xrightarrow{act} s' \wedge MB(acts, s')$

$$\begin{array}{ll}
final(middle, middle, true) & \\
(x, y, false) \xrightarrow{walk} (y, y, false) \quad \Leftarrow \quad x \not\equiv y & \\
(x, x, false) \xrightarrow{push} (middle, middle, false) \quad \Leftarrow \quad x \not\equiv middle & \\
(middle, middle, false) \xrightarrow{climb} (middle, middle, true) & \\
\textsf{standard inequality axioms} & \text{(see [75], Section 4)}
\end{array}$$

For a given state s, an expansion of $MB(acts, s)$ terminates because (1) final states are reachable from s, (2) only finitely many states are reachable and (3) each of them is reached at most once, i.e. $\longrightarrow$ does not run into cycles.

Bottling water. You have two empty bottles, one for three and one for five gallons of water and an unbounded water supply, and want to fill them with exactly four gallons. The bottles are in state (x, y) iff x and y are the numbers of gallons in the first resp. second bottle. A plan whose execution leads to the desired state is obtained by expanding the goal $bottles(acts, (0, 0))$ with axioms of the following specification.

BOTTLES = LIST `then`

$$\begin{array}{ll}
\texttt{sorts} & com \quad state = nat \times nat \\
\texttt{constructs} & fill1, fill2, empty1, empty2, 1to2, 2to1 :\rightarrow com \\
\texttt{static preds} & bottles : list(com) \times state \\
 & final : state \\
 & _ \xrightarrow{_} _ : state \times com \times state \\
\texttt{vars} & x, y : nat \quad act : com \quad acts : list(com) \quad s, s' : state \\
\texttt{Horn axioms} & bottles([], s) \quad \Leftarrow \quad final(s) \\
 & bottles(act : acts, s) \quad \Leftarrow \quad s \xrightarrow{act} s' \wedge bottles(acts, s') \\
 & final(x, y) \quad \Leftarrow \quad x + y \equiv 4 \\
 & (x, y) \xrightarrow{fill1} (3, y) \\
 & (x, y) \xrightarrow{fill2} (x, 5) \\
 & (x, y) \xrightarrow{empty1} (0, y) \\
 & (x, y) \xrightarrow{empty2} (x, 0) \\
 & (x, y) \xrightarrow{1to2} (0, x + y) \quad \Leftarrow \quad x + y \leq 5 \\
 & (x, y) \xrightarrow{1to2} (x + y - 5, 5) \quad \Leftarrow \quad x + y > 5 \\
 & (x, y) \xrightarrow{2to1} (x + y, 0) \quad \Leftarrow \quad x + y \leq 3 \\
 & (x, y) \xrightarrow{2to1} (3, x + y - 3) \quad \Leftarrow \quad x + y > 3
\end{array}$$

For a given state s, final states are reachable from s and only finitely many states are reachable. To ensure that expansions of the goal $bottles(acts, (0, 0))$ terminate we also require that each reachable state is achieved at most once. This is accomplished by accumulating visited states and expanding $bottles'(acts, (0, 0), \emptyset)$ instead of $bottles(acts, (0, 0))$:

BOTTLES' = BOTTLES `and` SET `then`

$$\begin{array}{ll}
\texttt{static preds} & bottles' : list(com) \times state \times set(state) \\
\texttt{vars} & act : com \quad acts : list(com) \quad s, s' : state \quad states : set(state) \\
\texttt{Horn axioms} & bottles'([], s, states) \quad \Leftarrow \quad final(s) \\
 & bottles'(act : acts, s, states) \\
 & \qquad \Leftarrow \quad s \xrightarrow{act} s' \wedge s' \notin states \wedge bottles'(acts, s', states \cup \{s'\})
\end{array}$$

The towers of Hanoi. A pile of circular blocks (represented as natural numbers) is to be carried from place A via place B to place C. The blocks are always piled up in the order of decreasing diameter. They are in state (x, y, z) iff x, y resp. z is the pile of blocks at place A, B resp. C.

TOWERS = LIST `then`

`sorts`	$place \quad com$
`hidsorts`	$pile = list(nat) \quad state = pile \times pile \times pile$
`constructs`	$A, B, C :\to place$
	$\_to\_ : place \times place \to com$
`static preds`	$towers : list(com) \times state$
	$admissable : nat \times list(nat)$
	$\_ \xrightarrow{-} \_ : state \times com \times state$
	$final : state$
`vars`	$x, y : nat \quad act : com \quad acts : list(com) \quad s, s' : state \quad a, b, c : pile$
`Horn axioms`	$towers([], s) \quad \Leftarrow \quad final(s)$
	$towers(act : acts, s) \quad \Leftarrow \quad s \xrightarrow{act} s' \wedge towers(acts, s')$
	$final([], [], c)$
	$(x : a, b, c) \xrightarrow{A\ to\ B} (a, x : b, c) \quad \Leftarrow \quad admissable(x, b)$
	$(x : a, b, c) \xrightarrow{A\ to\ C} (a, b, x : c) \quad \Leftarrow \quad admissable(x, c)$
	$(a, x : b, c) \xrightarrow{B\ to\ C} (a, b, x : c) \quad \Leftarrow \quad admissable(x, c)$
	$(a, x : b, c) \xrightarrow{B\ to\ A} (x : a, b, c) \quad \Leftarrow \quad admissable(x, a)$
	$(a, b, x : c) \xrightarrow{C\ to\ A} (x : a, b, c) \quad \Leftarrow \quad admissable(x, a)$
	$(a, b, x : c) \xrightarrow{C\ to\ B} (a, x : b, c) \quad \Leftarrow \quad admissable(x, b)$
	$admissable(x, [])$
	$admissable(x, y : a) \quad \Leftarrow \quad x \leq y$

For a given state s, final states are reachable from s and only finitely many states are reachable. To ensure that expansions of $towers(acts, s)$ terminate we also require that each reachable state is reached at most once. Analogously to BOTTLES', this is achieved by storing visited states.

The following well-known function, which is often used for introducing recursion, computes a plan for carrying blocks from A via B to C deterministically:

RECTOWERS = TOWERS `then`

`defuncts`	$plan : pile \times place \times place \times place \to list(com)$
`vars`	$x : nat \quad a : pile \quad source, aux, target : place$
`Horn axioms`	$plan([], source, aux, target) \equiv []$
	$plan(x : a, source, aux, target) \equiv plan(a, source, target, aux) +\!\!+ [source\ to\ target] +\!\!+ plan(a, aux, source, target)$

Exercise. Show that $plan$ is correct, i.e. prove that

$$sorted(a) \quad \Rightarrow \quad towers(plan(a, A, B, C), (a, [], [])) \tag{1}$$

is valid in the Herbrand model of RECTOWERS! Can the axioms for $plan$ be derived from an inductive proof of (a generalization of) (1) (cf. [76], Section 9; [72], Section 5.5)?

3.4 Lift controller

The "lift problem" is one of the benchmark examples for illustrating approaches to the specification of reactive systems (see, e.g., [13, 100]). People send stop requests from the inside or the outside of an elevator to a controller that causes the lift to move to the requested levels. Each lift (controller) state s is given by four destructors:

- $dir(s)$ yields the direction in which the lift is currently moving.
- $nextFloor(s)$ denotes the next floor a moving lift heads to.
- $requests(up, s)$ and $requests(down, s)$ are sorted lists of numbers representing the floors still to be served, where a stop request is still pending. The lists are updated by the functions *insert* and *tail*.
- $visits(s)$ returns the list of numbers denoting all visited floors up to the state s where the lift stops and meets a stop request. The order of $visits(s)$ agrees with the order in which the lift arrived at the corresponding floors.

A state s is built up of constructors representing the "history" of actions leading from the initial state *new* to s. The basic actions are $goto(n)$: a stop at floor n is requested; *stop*: the lift stops at the next floor in the direction in which it currently moves; *start*: the lift starts going; *pass*: the lift passes the next floor in the direction in which it currently moves, but does not stop there.

LIFT = LIST `then`

`sorts`	$two \quad act \quad state$
`objconstructs`	$up, down :\to two$
	$\tau :\to act$
	$goto, visit : nat \to act$
	$new :\to state$
	$goto : nat \times state \to state$
	$stop, start, pass : state \to state$
`destructs`	$dir : state \to two$
	$nextFloor : state \to nat$
	$requests : two \times state \to list(nat)$
	$apply : ((nat \times nat) \to bool) \times (nat \times nat) \to bool$
`defuncts`	$tail : list(nat) \to list(nat)$
	$incr : nat \times two \to nat$
	$turn : two \to two$
	$insert : nat \times list \times ((nat \times nat) \to bool) \to list$
	$gt, lt : nat \times nat \to bool$
`dynamic preds`	$\_ \stackrel{-}{\longrightarrow} \_ : state \times act \times state$
`static preds`	$no\_visit, no\_request : act$
`ν-preds`	$will\_be\_served, was\_requested : nat \times state$
`vars`	$d : two \quad s, s' : state \quad a : act \quad m, n : nat \quad L : list(nat) \quad \alpha : act \quad g : nat \times nat \to bool$
`Horn axioms`	$incr(n, up) \equiv n + 1 \qquad incr(n, down) \equiv n - 1$
	$turn(up) \equiv down \qquad turn(down) \equiv up$
	$tail([]) \equiv [] \qquad tail(n : L) \equiv L$
	$insert(m, n : L, g) \equiv m : n : L \quad \Leftarrow \quad g(m, n) \equiv true$
	$insert(n, n : L, g) \equiv n : L$
	$insert(m, n : L, g) \equiv n : insert(m, L) \quad \Leftarrow \quad g(m, n) \equiv false$
	$gt(m, n) \equiv true \quad \Leftarrow \quad m > n$
	$gt(m, n) \equiv false \quad \Leftarrow \quad m \leq n$
	$lt(m, n) \equiv gt(n, m)$
	$dir(new) \equiv up$
	$dir(goto(n, s)) \equiv dir(s)$
	$dir(stop(s)) \equiv dir(s)$
	$dir(start(s)) \equiv turn(dir(s)) \quad \Leftarrow \quad requests(dir(s), s) \equiv []$
	$dir(start(s)) \equiv dir(s) \quad \Leftarrow \quad requests(dir(s), s) \equiv n : L$

$dir(pass(s)) \equiv dir(s)$
$nextFloor(new) \equiv 1$
$nextFloor(goto(n,s)) \equiv nextFloor(s)$
$nextFloor(stop(s)) \equiv nextFloor(s)$
$nextFloor(start(s)) \equiv incr(nextFloor(s), turn(dir(s))) \quad \Leftarrow \quad requests(dir(s),s) \equiv []$
$nextFloor(start(s)) \equiv incr(nextFloor(s), dir(s)) \quad \Leftarrow \quad requests(dir(s),s) \equiv n:L$
$nextFloor(pass(s)) \equiv incr(nextFloor(s), dir(s))$
$requests(d,new) \equiv []$
$requests(up,goto(n,s)) \equiv insert(n, requests(up,s), gt) \quad \Leftarrow \quad n \geq nextFloor(s)$
$requests(up,goto(n,s)) \equiv insert(n, requests(down,s), lt) \quad \Leftarrow \quad n < nextFloor(s)$
$requests(down,goto(n,s)) \equiv insert(n, requests(down,s), lt) \quad \Leftarrow \quad n \leq nextFloor(s)$
$requests(down,goto(n,s)) \equiv insert(n, requests(up,s), gt) \quad \Leftarrow \quad n > nextFloor(s)$
$requests(d,stop(s)) \equiv tail(requests(d,s))$
$requests(d,start(s)) \equiv requests(d,s)$
$requests(d,pass(s)) \equiv requests(d,s)$
$s \xrightarrow{goto(n)} goto(n,s)$
$s \xrightarrow{visit(n)} stop(s) \quad \Leftarrow \quad requests(dir(s),s) \equiv n:L \wedge n \equiv nextFloor(s)$
$s \xrightarrow{\tau} start(s) \quad \Leftarrow \quad requests(up,s) \equiv n:L \vee requests(down,s) \equiv n:L$
$s \xrightarrow{\tau} pass(s) \quad \Leftarrow \quad requests(dir(s),s) \equiv n:L \wedge n \not\equiv nextFloor(s)$
$no\_visit(\tau) \qquad no\_request(\tau)$
$no\_visit(goto(n)) \qquad no\_request(visit(n))$

`co-Horn axioms` $will\_be\_served(n,s)$
$\Rightarrow \exists s' : s \xrightarrow{visit(n)} s' \vee \exists\, a, s' : (s \xrightarrow{a} s' \wedge no\_visit(a) \wedge will\_be\_served(n,s'))$
$was\_requested(n,s)$
$\Rightarrow \exists s' : s' \xrightarrow{goto(n)} s \vee \exists\, a, s' : (s' \xrightarrow{a} s \wedge no\_request(a) \wedge was\_requested(n,s'))$

Exercise. Show that the following formulas are valid in the Herbrand model of LIFT!

(1) $was\_requested(n,s) \Rightarrow will\_be\_served(n,s)$ (each request to visit a floor is served eventually).

(2) $will\_be\_served(n,s) \Rightarrow was\_requested(n,s)$ (a floor is visited only if it was requested for).

(3) $s' \xrightarrow{a} s \wedge dir(s') \equiv turn(dir(s)) \Rightarrow requests(dir(s),s) \equiv []$ (the lift changes its direction only if there are no pending requests in the current direction).

(4) $sorted(rev(requests(down,s)) +\!\!+ (nextFloor(s) : requests(up,s)))$ (the request lists are sorted upwards resp. downwards and their elements are greater resp. smaller than or equal to $nextFloor(s)$).

3.5 A command language with communication

We specify a simple imperative language that admits iteration, nondeterminism and broadcast communication via a shared queue. The basic schema of the specification is similar to ACCOUNT (cf. Section 3.1). Moreover, a hidden sort *environment* denotes sets of both "user stores" and queue states. A similar scenario is described by the *dynamic data type* SYSTEM of [5]. We presuppose a visible specification INT of integer arithmetic and regard *int* and *var* as subsorts of *exp* and *bool* as a subsort of *boolexp*. Actualizations of LIST and MAP (cf. Section 2.2) are used for specifying the queue and user stores. The empty sorts *prid* and *var* denote sets of process identifiers and program variables, respectively.

COM = ENTRY(*var*) `and` ENTRY(*prid*) `and` INT `and` LIST `and` MAP `and` SET `then`
`sorts` *exp boolexp com bufcom*

`hidsorts` $store \quad queue = list(int) \quad environment = set(store) \times queue$

`consts` $new : prid \rightarrow store$

$upd : int \times var \times store \rightarrow store$

$\_ := \_ : var \times exp \rightarrow com$

$\_? : boolexp \rightarrow com$

$send : exp \rightarrow com$

$receive : var \rightarrow com$

$\_;\_ : com \times com \rightarrow com$

$\_ + \_ : com \times com \rightarrow com$

$\_^* : com \rightarrow com$

$put, get : int \rightarrow bufcom$

$\_/\_ : com \times bufcom \rightarrow com$

$create : prid \rightarrow com$

$\_.\_ : prid \times com \rightarrow com$

$\_ : var \rightarrow exp$

$\_ : int \rightarrow exp$

$\_ : bool \rightarrow boolexp$

$add : exp \times exp \rightarrow exp$

$equal, greater : exp \times exp \rightarrow boolexp$

$Not : boolexp \rightarrow boolexp$

$And : boolexp \times boolexp \rightarrow boolexp$

`destructs` $owner : store \rightarrow prid$

$contents : store \rightarrow map(index, var)$

`defuncts` $skip, fail : \rightarrow com$

$if\_then\_else\_ : boolexp \times com \times com \rightarrow com$

$while\_do\_ : boolexp \times com \rightarrow com$

$if\_in\_then\_else\_ : boolexp \times prid \times com \times com \rightarrow com$

$while\_in\_do\_ : boolexp \times prid \times com \rightarrow com$

$eval : exp \rightarrow int$

$eval : boolexp \rightarrow bool$

$dequeue : queue \rightarrow queue$

$first : queue \rightarrow 1 + int$

`dynamic preds` $\_ \stackrel{\_}{\longrightarrow} \_ : queue \times com \times queue$

$\_ \stackrel{\_}{\longrightarrow} \_ : store \times com \times store$

$\_ \stackrel{\_}{\Longrightarrow} \_ : set(store) \times com \times set(store)$

$\_ \stackrel{\_}{\Longrightarrow} \_ : environment \times com \times environment$

ν`-preds` $\_ \approx \_ : com \times com$

`vars` $i, k : int \quad b : bool \quad x : var \quad p : prid \quad e, e' : exp \quad be, be' : boolexp \quad c, c' : com \quad bc : bufcom$

$s, s', s_1 : store \quad ss, ss' : set(store) \quad L, L' : queue$

$env, env', env_1, env_2 : environment$

`Horn axioms` $owner(new(p)) \equiv p$

$contents(new(p)) \equiv new$

$owner(upd(i,x,s)) \equiv owner(s)$

$contents(upd(i,x,s)) \equiv upd(i,x,contents(s))$

$L \stackrel{put(i)}{\longrightarrow} i : L$

$L \stackrel{get(i)}{\longrightarrow} dequeue(L) \quad \Leftarrow \quad first(L) \equiv (i)$

$s \stackrel{x:=e}{\longrightarrow} upd(i,x,s) \quad \Leftarrow \quad eval(s,e) \equiv i$

$$s \xrightarrow{be?} s \quad \Leftarrow \quad eval(s,be) \equiv true$$
$$s \xrightarrow{send(e)/put(i)} s \quad \Leftarrow \quad eval(s,e) \equiv i$$
$$s \xrightarrow{receive(x)/get(i)} upd(i,x,s)$$
$$s \xrightarrow{c;c'} s' \quad \Leftarrow \quad s \xrightarrow{c} s_1 \wedge s_1 \xrightarrow{c'} s'$$
$$s \xrightarrow{c+c'} s' \quad \Leftarrow \quad s \xrightarrow{c} s'$$
$$s \xrightarrow{c+c'} s' \quad \Leftarrow \quad s \xrightarrow{c'} s'$$
$$s \xrightarrow{c^*} s$$
$$s \xrightarrow{c^*} s' \quad \Leftarrow \quad s \xrightarrow{c;c^*} s'$$
$$ss \xRightarrow{create(p)} ss \cup \{new(p)\}$$
$$ss \xRightarrow{p.c} ss \setminus \{s\} \cup \{s'\} \quad \Leftarrow \quad s \xrightarrow{c} s' \ \wedge \ owner(s) \equiv p$$
$$ss \xRightarrow{p.c/bc} ss \setminus \{s\} \cup \{s'\} \quad \Leftarrow \quad s \xrightarrow{c/bc} s' \ \wedge \ owner(s) \equiv p$$
$$(ss,L) \xRightarrow{c} (ss',L) \quad \Leftarrow \quad ss \xRightarrow{c} ss'$$
$$(ss,L) \xRightarrow{c} (ss',L') \quad \Leftarrow \quad ss \xRightarrow{c/bc} ss' \ \wedge \ L \xrightarrow{bc} L'$$
$$env \xRightarrow{c;c'} env' \quad \Leftarrow \quad env \xRightarrow{c} env_1 \wedge env_1 \xRightarrow{c'} env'$$
$$env \xRightarrow{c+c'} env' \quad \Leftarrow \quad env \xRightarrow{c} env'$$
$$env \xRightarrow{c+c'} env' \quad \Leftarrow \quad env \xRightarrow{c'} env'$$
$$env \xRightarrow{c^*} env$$
$$env \xRightarrow{c^*} env' \quad \Leftarrow \quad env \xRightarrow{c;c^*} env'$$
$$skip \equiv true?$$
$$fail \equiv false?$$
$$if\ be\ then\ c\ else\ c' \ \equiv \ (be?;c) + (Not(be)?;c')$$
$$while\ be\ do\ c \ \equiv \ (be?;c)^*; Not(be)?$$
$$if\ be\ in\ p\ then\ c\ else\ c' \ \equiv \ (p.be?;c) + (p.Not(be)?;c')$$
$$while\ be\ in\ p\ do\ c \ \equiv \ (p.be?;c)^*; p.Not(be)?$$
$$eval(s,x) \equiv get(contents(s),x)$$
$$eval(s,i) \equiv i$$
$$eval(s,add(e,e')) \equiv eval(s,e) + eval(s,e')$$
$$eval(s,b) \equiv b$$
$$eval(s,equal(e,e')) \equiv true \quad \Leftarrow \quad eval(s,e) \equiv eval(s,e')$$
$$eval(s,equal(e,e')) \equiv false \quad \Leftarrow \quad eval(s,e) \not\equiv eval(s,e')$$
$$eval(s,greater(e,e')) \equiv true \quad \Leftarrow \quad eval(s,e) > eval(s,e')$$
$$eval(s,greater(e,e')) \equiv false \quad \Leftarrow \quad eval(s,e) \leq eval(s,e')$$
$$eval(s,Not(be)) \equiv not(eval(s,be))$$
$$eval(s,And(be,be')) \equiv (eval(s,be)\ and\ eval(s,be'))$$
$$dequeue([]) \equiv []$$
$$dequeue(i:[]) \equiv []$$
$$dequeue(i:k:L) \equiv i : dequeue(k:L)$$
$$first([]) \equiv ()$$
$$first(i:[]) \equiv (i)$$
$$first(i:k:L) \equiv first(k:L)$$

`co-Horn axioms`
$$c \approx c' \quad \Rightarrow \quad (env \xRightarrow{c} env_1 \ \Rightarrow \ \exists\ env_2 : (env \xRightarrow{c'} env_2 \wedge env_1 \sim env_2))$$
$$c \approx c' \quad \Rightarrow \quad (env \xRightarrow{c'} env_2 \ \Rightarrow \ \exists\ env_1 : (env \xRightarrow{c} env_1 \wedge env_1 \sim env_2))$$

Floyd-Hoare program assertions can be presented as first-order formulas over COM. For instance, let s, s' be *sort*-variables and c be a *com*-term. Then the formula

$$pre(s) \wedge s \xrightarrow{c} s' \quad \Rightarrow \quad post(s')$$

expresses the correctness the program presented by c w.r.t. the input/output relation given by *pre*/*post*: if the precondition *pre* holds true in s and if c transforms s into s', then s' satisfies the postcondition *post*. The classical rules of the **Hoare calculus** for proving assertions about sequential programs become expansion rules that are sound w.r.t. the Herbrand model of COM (cf. [75]):

assignment rule

$$\frac{pre(s) \wedge s \xrightarrow{x:=e} s' \quad \Rightarrow \quad post(s')}{pre(s) \quad \Rightarrow \quad post(s)[e/x]} \Uparrow$$

sequence rule

$$\frac{pre(s) \wedge s \xrightarrow{c;c'} s' \quad \Rightarrow \quad post(s')}{pre(s) \wedge s \xrightarrow{c} s' \quad \Rightarrow \quad q(s'), \qquad q(s) \wedge s \xrightarrow{c'} s' \quad \Rightarrow \quad post(s')} \Uparrow$$

conditional rule

$$\frac{pre(s) \wedge s \xrightarrow{if\ be\ then\ c\ else\ c'} s' \quad \Rightarrow \quad post(s')}{\begin{array}{c} pre(s) \wedge eval(s, be) = true \wedge s \xrightarrow{c} s' \quad \Rightarrow \quad post(s'), \\ pre(s) \wedge eval(s, be) = false \wedge s \xrightarrow{c'} s' \quad \Rightarrow \quad post(s') \end{array}} \Uparrow$$

loop rule

$$\frac{pre(s) \wedge s \xrightarrow{while\ be\ do\ c} s' \quad \Rightarrow \quad post(s')}{\begin{array}{c} pre(s) \quad \Rightarrow \quad inv(s), \\ inv(s) \wedge eval(s, be) = true \wedge s \xrightarrow{c} s' \quad \Rightarrow \quad inv(s'), \\ inv(s) \wedge eval(s, be) = false \quad \Rightarrow \quad post(s) \end{array}} \Uparrow$$

inv is usually called a **Hoare invariant**. Besides its occurrence in the loop rule it may support the proof that a loop terminates:

termination rule

$$\frac{pre(s) \quad \Rightarrow \quad s \xrightarrow{while\ be\ do\ c} s'}{\begin{array}{c} pre(s) \quad \Rightarrow \quad inv(s), \\ inv(s) \wedge eval(s, be) = true \wedge s \xrightarrow{c} s' \quad \Rightarrow \quad s \gg s' \wedge inv(s') \end{array}} \Uparrow$$

if $\gg$: *store* $\times$ *store* has a well-founded interpretation in the Herbrand model of COM

The **weakest (liberal) precondition of** $(c, post)$ can be specified by the following generalized Horn clauses:

$$wp(s) \quad \Leftarrow \quad wlp(s) \wedge s \xrightarrow{c} s' \qquad \text{resp.} \qquad wlp(s) \quad \Leftarrow \quad \forall s' : (s \xrightarrow{c} s' \Rightarrow post(s')).$$

4 Streams and processes

4.1 Streams

Infinite sequences are specified as follows.

STREAM = LIST `and` ENTRY($entry'$) `then`

`hidsorts`	$stream = stream(entry)$ $\;$ $stream' = stream(entry')$	
`constructs`	$\_\&\_ : entry \times stream \to stream'$	
	$blink :\to stream(nat)$	
	$nats : nat \to stream(nat)$	
	$iter_1, iter_2 : (entry \to entry) \times entry \to stream(entry)$	
	$rev : stream(bool) \to stream(bool)$	
	$odds : stream \to stream$	
	$zip : stream \times stream \to stream$	
	$map : (entry \to entry') \to (stream \to stream')$	
`destructs`	$head : stream \to entry$	
	$tail : stream \to stream$	
`defuncts`	$switch : nat \to nat$	
	$\_\#\_ : list \times stream \to stream$	
	$evens : stream \to stream$	
	$firstn : nat \times stream \to list$	
	$nthtail : nat \times stream \to stream$	
	$loop : (entry \to entry) \to nat \to entry$	
`preds`	$exists : (entry \to bool) \times stream$	
	$Nat : nat$	
`copreds`	$forall, forallExists : (entry \to bool) \times stream$	
	$fair : (entry \to bool) \times stream$	
`vars`	$n : nat$ $\;$ $x, y : entry$ $\;$ $L : list$ $\;$ $s, s' : stream$ $\;$ $bs : stream(bool)$	
	$f : entry \to entry'$ $\;$ $g : entry \to bool$	
`Horn axioms`	$head(x\&s) \equiv x$	$tail(x\&s) \equiv s$
	$head(blink) \equiv 0$	$tail(blink) \equiv 1\&blink$
	$head(nats(n)) \equiv n$	$tail(nats(n)) \equiv nats(suc(n))$
	$head(iter_1(f, x)) \equiv x$	$tail(iter_1(f, x)) \equiv iter_1(f, f(x))$
	$head(iter_2(f, x)) \equiv x$	$tail(iter_2(f, x)) \equiv map(f)(iter_2(f, x))$
	$head(zip(s, s')) \equiv head(s)$	$tail(zip(s, s')) \equiv zip(s', tail(s))$
	$head(rev(bs)) \equiv not(head(bs))$	$tail(rev(s)) \equiv rev(tail(s))$
	$head(odds(s)) \equiv head(s)$	$tail(odds(s)) \equiv odds(tail(tail(s)))$
	$head(map(f)(s)) \equiv f(head(s))$	$tail(map(f)(s)) \equiv map(f)(tail(s))$
	$switch(0) \equiv 1$	
	$switch(1) \equiv 0$	
	$nil\#s \equiv s$	
	$(x : L)\#s \equiv x\&(L\#s)$	
	$evens(s) \equiv odds(tail(s))$	
	$firstn(0, s) \equiv nil$	
	$firstn(suc(n), s) \equiv head(s) : firstn(n, tail(s))$	
	$nthtail(0, s) \equiv s$	
	$nthtail(suc(n), s) \equiv nthtail(n, tail(s))$	

$$
\begin{array}{ll}
 & loop(f)(0)(x) \equiv x \\
 & loop(f)(suc(n))(x) \equiv f(loop(f)(n)(x)) \\
 & exists(g,s) \quad \Leftarrow \quad g(head(s)) \equiv true \\
 & exists(g,s) \quad \Leftarrow \quad exists(g,tail(s)) \\
 & Nat(0) \\
 & Nat(suc(n)) \quad \Leftarrow \quad Nat(n) \\
\texttt{co-Horn axioms} & forall(g,s) \quad \Rightarrow \quad g(head(s)) \equiv true \wedge forall(g,tail(s)) \\
 & forallExists(g,s) \quad \Rightarrow \quad exists(g,s) \wedge forallExists(g,tail(s)) \\
\text{(A)} & fair(g,s) \ \Rightarrow \ forallExists(g,s) \\
\text{(B)} & fair(g,s) \ \Rightarrow \ exists(g,s) \wedge fair(g,tail(s)) \\
\text{(C)} & fair(g,s) \ \Rightarrow \ \exists n,s' : (forall(not \circ g, firstn(n,s)) \wedge nthtail(n,s) \equiv s' \wedge head(s') \equiv x \\
 & \qquad\qquad \wedge\ g(x) \equiv true \wedge fair(g,tail(s')))
\end{array}
$$

The final model (cf. [79], Section 6.5) interprets STREAM as follows. The *stream*-carrier consists of all infinite sequences over a set of entries (cf. [79], Ex. 2.5.7). & appends an entry to a stream. *blink* denotes a stream whose elements alternate between zeros and ones. $nats(n)$ generates the stream of all numbers starting from n. $odds(s)$ returns the stream of all elements of s that have odd-numbered positions in s. *zip* merges two streams into a single stream by alternatively appending an element of one stream to an element of the other stream. # concatenates a list and a stream into a stream. *head*, *tail*, *firstn*, *nthtail*, *map*, *exists*, *forall* and *forallExists* are the stream counterparts of the synonymous list functions. $fair(g,s)$ holds true iff s contains infinitely many elements satisfying g. Axioms A,B,C are pairwise inductively equivalent.

In the sequel, we show that the following behavioral equations are valid in Fin(STREAM):

$$rev(rev(s)) \sim s \tag{1}$$

$$odds(x\&s) \sim x\&evens(s) \tag{2}$$

$$evens(zip(s,s')) \sim s' \tag{3}$$

$$zip(odds(s),evens(s)) \sim s \tag{4}$$

$$iter_1(f,x) \sim iter_2(f,x) \tag{5}$$

The proofs are generated by *Expander 2* almost automatically. The main rules employed are coinduction, explicit induction, narrowing and resolution (cf. [78], [80, 81]). Alternative proofs using more specialized proof methods can be found in [91, 92], [20], [21] and [31], respectively.

`All`, `Any`, `&` and `|` denote ∀, ∃, ∧ and ∨, respectively.

Proof of (1):

```
rev(rev(S)) ~ S

Coinduction w.r.t.  s0 ~ s'0 ===> head(s0) = head(s'0) & tail(s0) ~ tail(s'0)
applied to the preceding formula leads to the formula

All s0 s'0:
(s0 = rev(rev(s'0)) ===> head(s0) = head(s'0) & tail(s0) = rev(rev(tail(s'0))))

Simplification applied to the preceding formula leads to the formula

All s'0: (head(rev(rev(s'0))) = head(s'0))
& All s'0: (tail(rev(rev(s'0))) = rev(rev(tail(s'0))))

The axiom  rev(s) = map(switch)(s)
applied at positions [1,0],[0,0] of the preceding formula leads to the formula
```

```
All s'0: (head(map(switch)(rev(s'0))) = head(s'0))
& All s'0: (tail(map(switch)(rev(s'0))) = rev(rev(tail(s'0))))

The axioms  tail(map(f)(s)) = map(f)(tail(s)) & head(map(f)(s)) = f(head(s))
applied at positions [1,0],[0,0] of the preceding formula leads to the formula

All s'0: (switch(head(rev(s'0))) = head(s'0))
& All s'0: (map(switch)(tail(rev(s'0))) = rev(rev(tail(s'0))))

The axiom  rev(s) = map(switch)(s)
applied at positions [1,0],[0,0] of the preceding formula leads to the formula

All s'0: (switch(head(map(switch)(s'0))) = head(s'0))
& All s'0: (map(switch)(tail(map(switch)(s'0))) = rev(rev(tail(s'0))))

The axioms  tail(map(f)(s)) = map(f)(tail(s)) & head(map(f)(s)) = f(head(s))
applied at positions [1,0],[0,0] of the preceding formula leads to the formula

All s'0: (switch(switch(head(s'0))) = head(s'0))
& All s'0: (map(switch)(map(switch)(tail(s'0))) = rev(rev(tail(s'0))))

The theorem  switch(switch(x)) = x
applied at position [0,0] of the preceding formula leads to the formula

All s'0: (head(s'0) = head(s'0))
& All s'0: (map(switch)(map(switch)(tail(s'0))) = rev(rev(tail(s'0))))

Simplification applied to the preceding formula leads to a new one. The current factor is given by

True

Simplification applied to the preceding formula leads to a new one. The current formula is given by

All s'0: (map(switch)(map(switch)(tail(s'0))) = rev(rev(tail(s'0))))

The axiom  rev(s) = map(switch)(s)
applied at position [0] of the preceding formula leads to the formula

All s'0: (map(switch)(map(switch)(tail(s'0))) = rev(map(switch)(tail(s'0))))

The axiom  rev(s) = map(switch)(s)
applied at position [0] of the preceding formula leads to the formula

All s'0: (map(switch)(map(switch)(tail(s'0))) = map(switch)(map(switch)(tail(s'0))))

Simplification applied to the entire formula leads to True.
```

Proof of (2):

```
odds(X:S) ~ X:evens(S)

The axiom  s ~ s' ===> head(s) = head(s') & tail(s) ~ tail(s')
applied at position [] of the preceding formula leads to the formula

head(odds(X:S)) = head(X:evens(S)) & tail(odds(X:S)) ~ tail(X:evens(S))

The axioms

tail(x:s) = s & head(x:s) = x
& head(odds(s)) = head(s) & tail(odds(s)) = odds(tail(tail(s)))
```

```
applied at positions [1],[1],[0],[0] of the preceding formula leads to the formula

head(X:S) = X & odds(tail(tail(X:S))) ~ evens(S)

The axioms  tail(x:s) = s & head(x:s) = x
applied at positions [1],[0] of the preceding formula leads to the formula

X = X & odds(tail(S)) ~ evens(S)

Simplification applied to the preceding formula leads to a new one. The current factor is given by

True

Simplification applied to the preceding formula leads to a new one. The current formula is given by

odds(tail(S)) ~ evens(S)

The axiom  evens(s) = odds(tail(s))
applied at position [] of the preceding formula leads to the formula

odds(tail(S)) ~ odds(tail(S))

Simplification applied to the entire formula leads to True.
```

Proof of (3):

```
evens(zip(S,S')) ~ S'

The axiom  evens(s) = odds(tail(s))
applied at position [] of the preceding formula leads to the formula

odds(tail(zip(S,S'))) ~ S'

The axiom  tail(zip(s,s')) = zip(s',tail(s))
applied at position [] of the preceding formula leads to the formula

odds(zip(S',tail(S))) ~ S'

Coinduction w.r.t.  s0 ~ s'0 ===> head(s0) = head(s'0) & tail(s0) ~ tail(s'0)
applied to the preceding formula leads to the formula

All s0 s'0:
(Any S0: (s0 = odds(zip(s'0,tail(S0))))
 ===> head(s0) = head(s'0) & Any S1: (tail(s0) = odds(zip(tail(s'0),tail(S1)))))

Simplification applied to the preceding formula leads to the formula

All s'0 S0: (head(odds(zip(s'0,tail(S0)))) = head(s'0))
& All s'0 S0: (Any S1: (tail(odds(zip(s'0,tail(S0)))) = odds(zip(tail(s'0),tail(S1)))))

The axiom  head(odds(s)) = head(s)
applied at position [0,0] of the preceding formula leads to the formula

All s'0 S0: (head(zip(s'0,tail(S0))) = head(s'0))
& All s'0 S0: (Any S1: (tail(odds(zip(s'0,tail(S0)))) = odds(zip(tail(s'0),tail(S1)))))

The axiom  head(zip(s,s')) = head(s)
applied at position [0,0] of the preceding formula leads to the formula

All s'0 S0: (head(s'0) = head(s'0))
```

```
& All s'0 S0: (Any S1: (tail(odds(zip(s'0,tail(S0)))) = odds(zip(tail(s'0),tail(S1)))))

Simplification applied to the preceding formula leads to a new one. The current factor is given by

True

Simplification applied to the preceding formula leads to a new one. The current formula is given by

All s'0 S0: (Any S1: (tail(odds(zip(s'0,tail(S0)))) = odds(zip(tail(s'0),tail(S1)))))

The axiom  tail(odds(s)) = odds(tail(tail(s)))
applied at position [0,0] of the preceding formula leads to the formula

All s'0 S0: (Any S1: (odds(tail(tail(zip(s'0,tail(S0))))) = odds(zip(tail(s'0),tail(S1)))))

The axiom  tail(zip(s,s')) = zip(s',tail(s))
applied at position [0,0] of the preceding formula leads to the formula

All s'0 S0: (Any S1: (odds(tail(zip(tail(S0),tail(s'0)))) = odds(zip(tail(s'0),tail(S1)))))

The axiom  tail(zip(s,s')) = zip(s',tail(s))
applied at position [0,0] of the preceding formula leads to the formula

All s'0 S0: (Any S1: (odds(zip(tail(s'0),tail(tail(S0)))) = odds(zip(tail(s'0),tail(S1)))))

Substituting tail(S0) for S1 applied at position [0] of the preceding formula leads to the formula

All s'0 S0: (odds(zip(tail(s'0),tail(tail(S0)))) = odds(zip(tail(s'0),tail(tail(S0)))))

Simplification applied to the entire formula leads to True.
```

Proof of (4):

```
zip(odds(S),evens(S)) ~ S

Coinduction w.r.t.  s0 ~ s'0 ===> head(s0) = head(s'0) & tail(s0) ~ tail(s'0)
applied to the preceding formula leads to the formula

All s0 s'0:
(s0 = zip(odds(s'0),evens(s'0))
 ===> head(s0) = head(s'0) & tail(s0) = zip(odds(tail(s'0)),evens(tail(s'0))))

Simplification applied to the preceding formula leads to the formula

All s'0: (head(zip(odds(s'0),evens(s'0))) = head(s'0))
& All s'0: (tail(zip(odds(s'0),evens(s'0))) = zip(odds(tail(s'0)),evens(tail(s'0))))

The axioms  tail(zip(s,s')) = zip(s',tail(s)) & head(zip(s,s')) = head(s)
applied at positions [1,0],[0,0] of the preceding formula leads to the formula

All s'0: (head(odds(s'0)) = head(s'0))
& All s'0: (zip(evens(s'0),tail(odds(s'0))) = zip(odds(tail(s'0)),evens(tail(s'0))))

The axioms  tail(odds(s)) = odds(tail(tail(s))) & head(odds(s)) = head(s)
applied at positions [1,0],[0,0] of the preceding formula leads to the formula

All s'0: (head(s'0) = head(s'0))
& All s'0: (zip(evens(s'0),odds(tail(tail(s'0)))) = zip(odds(tail(s'0)),evens(tail(s'0))))

Simplification applied to the preceding formula leads to a new one. The current factor is given by
```

```
True

Simplification applied to the preceding formula leads to a new one. The current formula is given by

All s'0: (zip(evens(s'0),odds(tail(tail(s'0)))) = zip(odds(tail(s'0)),evens(tail(s'0))))

The axiom  evens(s) = odds(tail(s))
applied at positions [0],[0] of the preceding formula leads to the formula

All s'0: (zip(odds(tail(s'0)),odds(tail(tail(s'0)))) = zip(odds(tail(s'0)),odds(tail(tail(s'0)))))

Simplification applied to the entire formula leads to True.
```

Proof of (5):

```
iter1(F,N) ~ iter2(F,N)

The theorem  s = map(loop(f)(0))(s)                                    (6)
applied at position [] of the preceding formula leads to the formula

Any f0: (iter1(F,N) ~ map(loop(f0)(0))(iter2(F,N)))

The theorem  map(loop(f)(n))(iter2(f,x)) = iter1(f,loop(f)(n)(x))      (7)
applied at position [0] of the preceding formula leads to the formula

Any f0: (iter1(f0,N) ~ iter1(f0,loop(f0)(0)(N)) & F = f0)

Simplification applied to the preceding formula leads to the formula

iter1(F,N) ~ iter1(F,loop(F)(0)(N))

The theorem  loop(f)(0)(x) = x
applied at position [] of the preceding formula leads to the formula

iter1(F,N) ~ iter1(F,N)

Simplification applied to the entire formula leads to True.
```

Proof of Lemma (6):

```
map(loop(F)(0))(S) ~ S

Coinduction w.r.t.  s0 ~ s'0 ===> head(s0) = head(s'0) & tail(s0) ~ tail(s'0)
applied to the preceding formula leads to the formula

All s0 s'0:
(Any F0: (s0 = map(loop(F0)(0))(s'0))
 ===> head(s0) = head(s'0) & Any F1: (tail(s0) = map(loop(F1)(0))(tail(s'0))))

Simplification applied to the preceding formula leads to the formula

All s'0 F0: (head(map(loop(F0)(0))(s'0)) = head(s'0))
& All s'0 F0: (Any F1: (tail(map(loop(F0)(0))(s'0)) = map(loop(F1)(0))(tail(s'0))))

The axioms  tail(map(f)(s)) = map(f)(tail(s)) & head(map(f)(s)) = f(head(s))
applied at positions [1,0,0],[0,0] of the preceding formula leads to the formula

All s'0 F0: (loop(F0)(0)(head(s'0)) = head(s'0))
& All s'0 F0: (Any F1: (map(loop(F0)(0))(tail(s'0)) = map(loop(F1)(0))(tail(s'0))))
```

```
The axiom  loop(f)(0)(x) = x
applied at position [0,0] of the preceding formula leads to the formula

All s'0 F0: (head(s'0) = head(s'0))
& All s'0 F0: (Any F1: (map(loop(F0)(0))(tail(s'0)) = map(loop(F1)(0))(tail(s'0))))

Simplification applied to the preceding formula leads to a new one. The current factor is given by

True

Simplification applied to the preceding formula leads to a new one. The current formula is given by

All s'0 F0: (Any F1: (map(loop(F0)(0))(tail(s'0)) = map(loop(F1)(0))(tail(s'0))))

Substituting F0 for F1 applied at position [0] of the preceding formula leads to the formula

All s'0 F0: (map(loop(F0)(0))(tail(s'0)) = map(loop(F0)(0))(tail(s'0)))

Simplification applied to the entire formula leads to True.
```

Proof of Lemma (7):

```
iter1(F,loop(F)(N)(X)) ~ map(loop(F)(N))(iter2(F,X))

Coinduction w.r.t.  s0 ~ s'0 ===> head(s0) = head(s'0) & tail(s0) ~ tail(s'0)
applied to the preceding formula leads to the formula

All s0 s'0:
(Any F0 N0 X0: (s0 = iter1(F0,loop(F0)(N0)(X0)) & s'0 = map(loop(F0)(N0))(iter2(F0,X0)))
 ===> head(s0) = head(s'0)
      & Any F1 N1 X1:
        (tail(s0) = iter1(F1,loop(F1)(N1)(X1)) & tail(s'0) = map(loop(F1)(N1))(iter2(F1,X1))))

Simplification applied to the preceding formula leads to the formula

All F0 N0 X0: (head(iter1(F0,loop(F0)(N0)(X0))) = head(map(loop(F0)(N0))(iter2(F0,X0))))
& All F0 N0 X0:
  (Any F1 N1 X1:
   (tail(iter1(F0,loop(F0)(N0)(X0))) = iter1(F1,loop(F1)(N1)(X1))
    & tail(map(loop(F0)(N0))(iter2(F0,X0))) = map(loop(F1)(N1))(iter2(F1,X1))))

Simplification applied to the preceding formula leads to a new one. The current factor is given by

All F0 N0 X0: (head(iter1(F0,loop(F0)(N0)(X0))) = head(map(loop(F0)(N0))(iter2(F0,X0))))

The axioms  head(map(f)(s)) = f(head(s)) & head(iter1(f,x)) = x
applied at positions [0],[1] of the preceding formula leads to the factor

All F0 N0 X0: (loop(F0)(N0)(X0) = loop(F0)(N0)(head(iter2(F0,X0))))

The axiom  head(iter2(f,x)) = x
applied at position [1,1] of the preceding formula leads to the factor

All F0 N0 X0: (loop(F0)(N0)(X0) = loop(F0)(N0)(X0))

Simplification applied to the preceding formula leads to the factor

True

Simplification applied to the preceding formula leads to a new one. The current formula is given by
```

```
All F0 N0 X0:
(Any F1 N1 X1:
 (tail(iter1(F0,loop(F0)(N0)(X0))) = iter1(F1,loop(F1)(N1)(X1))
  & tail(map(loop(F0)(N0))(iter2(F0,X0))) = map(loop(F1)(N1))(iter2(F1,X1))))

The axioms  tail(map(f)(s)) = map(f)(tail(s)) & tail(iter1(f,x)) = iter1(f,f(x))
applied at positions [0,0,0,0],[0,0,1,0] of the preceding formula leads to the formula

All F0 N0 X0:
(Any F1 N1 X1:
 (iter1(F0,F0(loop(F0)(N0)(X0))) = iter1(F1,loop(F1)(N1)(X1))
  & map(loop(F0)(N0))(tail(iter2(F0,X0))) = map(loop(F1)(N1))(iter2(F1,X1))))

The axiom  tail(iter2(f,x)) = map(f)(iter2(f,x))
applied at position [0,0,1,0,1] of the preceding formula leads to the formula

All F0 N0 X0:
(Any F1 N1 X1:
 (iter1(F0,F0(loop(F0)(N0)(X0))) = iter1(F1,loop(F1)(N1)(X1))
  & map(loop(F0)(N0))(map(F0)(iter2(F0,X0))) = map(loop(F1)(N1))(iter2(F1,X1))))

Substituting F0 for F1 applied at position [0] of the preceding formula leads to the formula

All F0 N0 X0:
(Any N1 X1:
 (iter1(F0,F0(loop(F0)(N0)(X0))) = iter1(F0,loop(F0)(N1)(X1))
  & map(loop(F0)(N0))(map(F0)(iter2(F0,X0))) = map(loop(F0)(N1))(iter2(F0,X1))))

The theorem  f(loop(f)(n)(x)) = loop(f)(suc(n))(x)
applied at position [0,0,0] of the preceding formula leads to the formula

All F0 N0 X0:
(Any N1 X1:
 (iter1(F0,loop(F0)(suc(N0))(X0)) = iter1(F0,loop(F0)(N1)(X1))
  & map(loop(F0)(N0))(map(F0)(iter2(F0,X0))) = map(loop(F0)(N1))(iter2(F0,X1))))

Substituting suc(N0) for N1 applied at position [0] of the preceding formula leads to the formula

All F0 N0 X0:
(Any X1:
 (iter1(F0,loop(F0)(suc(N0))(X0)) = iter1(F0,loop(F0)(suc(N0))(X1))
  & map(loop(F0)(N0))(map(F0)(iter2(F0,X0))) = map(loop(F0)(suc(N0)))(iter2(F0,X1))))

Substituting X0 for X1 applied at position [0] of the preceding formula leads to the formula

All F0 N0 X0:
(iter1(F0,loop(F0)(suc(N0))(X0)) = iter1(F0,loop(F0)(suc(N0))(X0))
 & map(loop(F0)(N0))(map(F0)(iter2(F0,X0))) = map(loop(F0)(suc(N0)))(iter2(F0,X0)))

Simplification applied to the preceding formula leads to the formula

All F0 N0 X0: (map(loop(F0)(N0))(map(F0)(iter2(F0,X0))) = map(loop(F0)(suc(N0)))(iter2(F0,X0)))

The theorem  map(loop(f)(n))(map(f)(s)) = map(loop(f)(suc(n)))(s)       (8)
applied at position [0] of the preceding formula leads to the formula

All F0 N0 X0: (map(loop(F0)(suc(N0)))(iter2(F0,X0)) = map(loop(F0)(suc(N0)))(iter2(F0,X0)))

Simplification applied to the entire formula leads to True.
```

Proof of Lemma (8):

```
map(loop(F)(suc(N)))(S) ~ map(loop(F)(N))(map(F)(S))

Coinduction w.r.t.  s0 ~ s'0 ===> head(s0) = head(s'0) & tail(s0) ~ tail(s'0)
applied to the preceding formula leads to the formula

All s0 s'0:
(Any F0 N0 S0: (s0 = map(loop(F0)(suc(N0)))(S0) & s'0 = map(loop(F0)(N0))(map(F0)(S0)))
 ===> head(s0) = head(s'0)
      & Any F1 N1 S1:
        (tail(s0) = map(loop(F1)(suc(N1)))(S1) & tail(s'0) = map(loop(F1)(N1))(map(F1)(S1))))

Simplification applied to the preceding formula leads to the formula

All F0 N0 S0: (head(map(loop(F0)(suc(N0)))(S0)) = head(map(loop(F0)(N0))(map(F0)(S0))))
& All F0 N0 S0:
  (Any F1 N1 S1:
   (tail(map(loop(F0)(suc(N0)))(S0)) = map(loop(F1)(suc(N1)))(S1)
    & tail(map(loop(F0)(N0))(map(F0)(S0))) = map(loop(F1)(N1))(map(F1)(S1))))

The axioms  tail(map(f)(s)) = map(f)(tail(s)) & head(map(f)(s)) = f(head(s))
applied at positions [0,0,0],[0,0,1],[1,0,0,0,0],[1,0,0,1,0] of the preceding formula leads to the formula

All F0 N0 S0: (loop(F0)(suc(N0))(head(S0)) = loop(F0)(N0)(head(map(F0)(S0))))
& All F0 N0 S0:
  (Any F1 N1 S1:
   (map(loop(F0)(suc(N0)))(tail(S0)) = map(loop(F1)(suc(N1)))(S1)
    & map(loop(F0)(N0))(tail(map(F0)(S0))) = map(loop(F1)(N1))(map(F1)(S1))))

Simplification applied to the preceding formula leads to a new one. The current factor is given by

All F0 N0 S0: (loop(F0)(suc(N0))(head(S0)) = loop(F0)(N0)(head(map(F0)(S0))))

The axiom  head(map(f)(s)) = f(head(s))
applied at position [1,1] of the preceding formula leads to the factor

All F0 N0 S0: (loop(F0)(suc(N0))(head(S0)) = loop(F0)(N0)(F0(head(S0))))

The theorem  loop(f)(suc(n))(x) = f(loop(f)(n)(x))
applied at position [] of the preceding formula leads to the factor

All F0 N0 S0: (F0(loop(F0)(N0)(head(S0))) = loop(F0)(N0)(F0(head(S0))))

The theorem  loop(f)(n)(f(x)) = f(loop(f)(n)(x))                        (9)
applied at position [] of the preceding formula leads to the factor

All F0 N0 S0: (F0(loop(F0)(N0)(head(S0))) = F0(loop(F0)(N0)(head(S0))))

Simplification applied to the preceding formula leads to the factor

True

Simplification applied to the preceding formula leads to a new one. The current formula is given by

All F0 N0 S0:
(Any F1 N1 S1:
 (map(loop(F0)(suc(N0)))(tail(S0)) = map(loop(F1)(suc(N1)))(S1)
  & map(loop(F0)(N0))(tail(map(F0)(S0))) = map(loop(F1)(N1))(map(F1)(S1))))

The axiom  tail(map(f)(s)) = map(f)(tail(s))
applied at position [0,0,1,0,1] of the preceding formula leads to the formula

All F0 N0 S0:
```

```
(Any F1 N1 S1:
 (map(loop(F0)(suc(N0)))(tail(S0)) = map(loop(F1)(suc(N1)))(S1)
  & map(loop(F0)(N0))(map(F0)(tail(S0))) = map(loop(F1)(N1))(map(F1)(S1))))

Substituting F0 for F1 applied at position [0] of the preceding formula leads to the formula

All F0 N0 S0:
(Any N1 S1:
 (map(loop(F0)(suc(N0)))(tail(S0)) = map(loop(F0)(suc(N1)))(S1)
  & map(loop(F0)(N0))(map(F0)(tail(S0))) = map(loop(F0)(N1))(map(F0)(S1))))

Substituting N0 for N1 applied at position [0] of the preceding formula leads to the formula

All F0 N0 S0:
(Any S1:
 (map(loop(F0)(suc(N0)))(tail(S0)) = map(loop(F0)(suc(N0)))(S1)
  & map(loop(F0)(N0))(map(F0)(tail(S0))) = map(loop(F0)(N0))(map(F0)(S1))))

Substituting tail(S0) for S1 applied at position [0] of the preceding formula leads to the formula

All F0 N0 S0:
(map(loop(F0)(suc(N0)))(tail(S0)) = map(loop(F0)(suc(N0)))(tail(S0))
 & map(loop(F0)(N0))(map(F0)(tail(S0))) = map(loop(F0)(N0))(map(F0)(tail(S0))))

Simplification applied to the entire formula leads to True.
```

Proof of Lemma (9):

```
Nat(N) ==> loop(F)(N)(F(X)) = F(loop(F)(N)(X))

Selecting induction variables applied at position [] of the preceding formula leads to the formula

All F X: (Nat(!N) ==> loop(F)(!N)(F(X)) = F(loop(F)(!N)(X)))

The axioms  loop(f)(0)(x) = x & loop(f)(suc(n))(x) = f(loop(f)(n)(x))
applied at position [0,1] of the preceding formula leads to the formula

All F X:
(Nat(!N)
 ==> F(X) = F(loop(F)(0)(X)) & !N = 0
     | Any n0: (F(loop(F)(n0)(F(X))) = F(loop(F)(suc(n0))(X)) & !N = suc(n0)))

The axioms  loop(f)(suc(n))(x) = f(loop(f)(n)(x)) & loop(f)(0)(x) = x
applied at positions [0,1,1,0,0],[0,1,0,0] of the preceding formula leads to the formula

All F X:
(Nat(!N)
 ==> F(X) = F(X) & !N = 0
     | Any n0: (F(loop(F)(n0)(F(X))) = F(F(loop(F)(n0)(X))) & !N = suc(n0)))

Simplification applied to the preceding formula leads to the formula

All F X:
(Nat(!N) ==> !N = 0 | Any n0: (F(loop(F)(n0)(F(X))) = F(F(loop(F)(n0)(X))) & !N = suc(n0)))

The theorem  loop(F)(N0)(F(X0)) = F(loop(F)(N0)(X0)) <=== Nat(N0) & !N >> N0
applied at position [0,1,1,0,0] of the preceding formula leads to the formula

All F X:
(Nat(!N)
 ==> !N = 0
```

```
    | Any n0: ((F(F(loop(F)(n0)(X))) = F(F(loop(F)(n0)(X))) & !N >> n0 & Nat(n0)) & !N = suc(n0)))

Simplification applied to the preceding formula leads to the formula

Nat(!N) ==> !N = 0 | Any n0: (!N >> n0 & Nat(n0) & !N = suc(n0))

The axiom  suc(x) >> x
applied at position [1,1,0,0] of the preceding formula leads to the formula

Nat(!N) ==> !N = 0 | Any n0: (!N = suc(n0) & Nat(n0) & !N = suc(n0))

The axioms  Nat(0) & (Nat(suc(x)) <=== Nat(x))
applied at position [0] of the preceding formula leads to the formula

!N = 0 | Any x0: (Nat(x0) & !N = suc(x0)) ==> !N = 0 | Any n0: (!N = suc(n0) & Nat(n0))

Simplification applied to the entire formula leads to True.
```

4.2 Finite and infinite sequences

For any correct actualization SP of STREAM that assigns the sort s to *entry*, $Fin(SP)_{stream}$ is embedded in $Ini(SP)_s^{\mathbb{N}}$ and thus in the final F-coalgebra where $F(A) =_{def} Ini(SP)_s \times A$ (see Section 6). Domains of finite *and* infinite streams can be specified either relationally in terms of a transition predicate $\longrightarrow: stream \times entry \times stream$, which replaces the destructors *head* and *tail* of STREAM, or functionally by combining *head* and *tail* to a single destructor $ht : stream \to 1 + (entry \times stream)$ (cf. Section 1). The latter solution "implements" the final G-coalgebra where $G(A) =_{def} 1 + (Ini(SP)_s \times A)$ analogously to the way HNAT implements the final F-coalgebra where $F(A) =_{def} 1 + A$ (see Sections 2.1 and 6).

RSTREAM = LIST `and` ENTRY($entry'$) `then`

`hidsorts`	$rstream = rstream(entry) \quad rstream' = rstream(entry')$
`constructs`	$nil :\to rstream$
	$\_\&\_ : (entry \to entry') \times rstream \to rstream'$
	$blink :\to rstream(nat)$
	$nats : nat \to rstream$
	$odds, evens : rstream \to rstream$
	$\_@\_ : rstream \times rstream \to rstream$
	$zip : rstream \times rstream \to rstream$
	$map : (entry \to entry) \times rstream \to rstream$
	$filter : (entry \to bool) \times rstream \to rstream$
`defuncts`	$firstn : nat \times rstream \to list$
	$nthtail : nat \times rstream \to rstream$
`destructors`	$isnil : rstream$
`transpreds`	$\_ \stackrel{\_}{\longrightarrow} \_ : rstream \times entry \times rstream$
`static preds`	$finite : rstream$
	$exists : (entry \to bool) \times rstream$
ν`-preds`	$infinite : rstream$
	$forall, forallExists : (entry \to bool) \times rstream$
	$fair : (entry \to bool) \times rstream$
`vars`	$n : nat \quad x, y : entry \quad L : list \quad s, s', t, t' : rstream \quad f : entry \to entry' \quad g : entry \to bool$
`Horn axioms`	$x\&s \stackrel{x}{\longrightarrow} s$
	$blink \stackrel{0}{\longrightarrow} 1\&blink$
	$nats(n) \stackrel{n}{\longrightarrow} nats(suc(n))$

$$
\begin{array}{ll}
& odds(s) \xrightarrow{x} odds(t) \quad \Leftarrow \quad s \xrightarrow{x} s' \wedge s' \xrightarrow{y} t \\
& evens(s) \xrightarrow{x} evens(t) \quad \Leftarrow \quad s \xrightarrow{y} s' \wedge s' \xrightarrow{x} t \\
& s@s' \xrightarrow{x} t@s' \quad \Leftarrow \quad s \xrightarrow{x} t \\
& s@s' \xrightarrow{x} s@t \quad \Leftarrow \quad isnil(s) \wedge s' \xrightarrow{x} t \\
& zip(s,s') \xrightarrow{x} zip(s',t) \quad \Leftarrow \quad s \xrightarrow{x} t \\
& zip(s,s') \xrightarrow{x} zip(s,t') \quad \Leftarrow \quad isnil(s) \wedge s' \xrightarrow{x} t' \\
& map(f,s) \xrightarrow{f(x)} map(f,t) \quad \Leftarrow \quad s \xrightarrow{x} t \\
& filter(g,s) \xrightarrow{x} filter(g,t) \quad \Leftarrow \quad s \xrightarrow{x} t \wedge g(x) \equiv true \\
& filter(g,s) \xrightarrow{y} t' \quad \Leftarrow \quad s \xrightarrow{x} t \wedge g(x) \equiv false \wedge filter(g,t) \xrightarrow{y} t' \\
& isnil(nil) \\
& isnil(odds(s)) \quad \Leftarrow \quad isnil(s) \\
& isnil(evens(s)) \quad \Leftarrow \quad isnil(s) \\
& isnil(evens(s)) \quad \Leftarrow \quad s \xrightarrow{x} t \wedge isnil(t) \\
& isnil(s@s') \quad \Leftarrow \quad isnil(s) \wedge isnil(s') \\
& isnil(zip(s,s')) \quad \Leftarrow \quad isnil(s) \wedge isnil(s') \\
& isnil(map(f,s)) \quad \Leftarrow \quad isnil(s) \\
& isnil(filter(g,s)) \quad \Leftarrow \quad isnil(s) \\
& isnil(filter(g,s)) \quad \Leftarrow \quad s \xrightarrow{x} t \wedge g(x) \equiv false \wedge isnil(filter(g,t)) \\
& finite(s) \quad \Leftarrow \quad isnil(s) \\
& finite(s) \quad \Leftarrow \quad s \xrightarrow{x} t \wedge isnil(t) \\
& exists(g,s) \quad \Leftarrow \quad s \xrightarrow{x} t \wedge g(x) \equiv true \\
& exists(g,s) \quad \Leftarrow \quad s \xrightarrow{x} t \wedge exists(g,t) \\
& firstn(0,s) \equiv nil \\
& firstn(suc(n),s) \equiv x : firstn(n,t) \quad \Leftarrow \quad s \xrightarrow{x} t \\
& nthtail(0,s) \equiv s \\
& nthtail(suc(n),s) \equiv nthtail(n,t) \quad \Leftarrow \quad s \xrightarrow{x} t \\
\texttt{co-Horn axioms} & infinite(s) \;\Rightarrow\; \exists x,t : (s \xrightarrow{x} t \wedge infinite(t)) \\
& forall(g,s) \;\Rightarrow\; (s \xrightarrow{x} t \;\Rightarrow\; (g(x) \equiv true \wedge forall(g,t))) \\
& forallExists(g,s) \;\Rightarrow\; exists(g,s) \\
& forallExists(g,s) \;\Rightarrow\; (s \xrightarrow{x} t \;\Rightarrow\; forallExists(g,t)) \\
\text{(A)} & fair(g,s) \;\Rightarrow\; forallExists(g,s) \\
\text{(B1)} & fair(g,s) \;\Rightarrow\; exists(g,s) \\
\text{(B2)} & fair(g,s) \;\Rightarrow\; (s \xrightarrow{x} t \;\Rightarrow\; fair(g,t)) \\
\text{(C)} & fair(g,s) \;\Rightarrow\; \exists n,s' : (forall(not \circ g, firstn(n,s)) \wedge nthtail(n,s) \equiv s' \wedge s' \xrightarrow{x} t \\
& \qquad\qquad \wedge\; g(x) \equiv true \wedge fair(g,t))
\end{array}
$$

In the final RSTREAM-model, $s \xrightarrow{x} t$ holds true if x is the first entry and t is the rest of s, @ is the concatenation of streams and $finite$ and $infinite$ distinguish finite from infinite streams. A comprehension function like $filter$ would not make sense within STREAM because it may return finite sequences. The other function symbols and predicates are interpreted as the synomymous symbols of STREAM.

The second specification of finite and infinite streams is built along the lines of HNAT because we also need to specify a partial destructor and thus to introduce a sum sort for incorporating "undefined" values. Similar to STREAM, but in contrast to RSTREAM, behavioral equivalence is induced by a *functional* observer.

COLIST = ENTRY($entry$) `and` ENTRY($entry'$) `and` NAT `then`

$$
\begin{array}{ll}
\texttt{hidsorts} & colist = colist(entry) \quad colist' = colist(entry') \\
\texttt{constructs} & nil :\to colist \\
& \_\&\_ : entry \times colist \to colist
\end{array}
$$

	$blink :\to colist(nat)$
	$nats : nat \to colist$
	$\_@\_ : colist \times colist \to colist$
	$zip : colist \times colist \to colist$
	$map : (entry \to entry) \times colist \to colist'$
`destructs`	$ht : colist \to 1 + (entry \times colist)$
`defuncts`	$evens : colist \to colist$
	$firstn : nat \times colist \to colist$
	$nthtail : nat \times colist \to 1 + colist$
`static preds`	$isnil : colist$
	$exists : (entry \to bool) \times colist$
ν`-preds`	$forall, forallExists : (entry \to bool) \times colist$
	$fair : (entry \to bool) \times colist$
	$infinite : colist$
`vars`	$n : nat \quad x, y : entry \quad s, s', t : colist \quad f : entry \to entry' \quad g : entry \to bool$
`Horn axioms`	$ht(nil) \equiv ()$
	$ht(x\&s) \equiv (x, s)$
	$ht(blink) \equiv (0, 1\&blink)$
	$ht(nats(n)) \equiv (n, nats(suc(n)))$
	$ht(s@s') \equiv ht(s') \quad \Leftarrow \quad ht(s) \equiv ()$
	$ht(s@s') \equiv (x, t@s') \quad \Leftarrow \quad ht(s) \equiv (x, t)$
	$ht(zip(s, s')) \equiv ht(s') \quad \Leftarrow \quad ht(s) \equiv ()$
	$ht(zip(s, s')) \equiv (x, zip(s', t)) \quad \Leftarrow \quad ht(s) \equiv (x, t)$
	$ht(map(f, s)) \equiv () \quad \Leftarrow \quad ht(s) \equiv ()$
	$ht(map(f, s)) \equiv (f(s), map(f, t)) \quad \Leftarrow \quad ht(s) \equiv (x, t)$
	$evens(s) \equiv () \quad \Leftarrow \quad ht(s) \equiv ()$
	$evens(s) \equiv odds(t) \quad \Leftarrow \quad ht(s) \equiv (x, t)$
	$firstn(0, s) \equiv nil$
	$firstn(suc(n), s) \equiv x\&firstn(n, t) \quad \Leftarrow \quad ht(s) \equiv (x, t)$
	$nthtail(n, s) \equiv nil \quad \Leftarrow \quad ht(s) \equiv ()$
	$nthtail(0, s) \equiv (s) \quad \Leftarrow \quad ht(s) \equiv (x, t)$
	$nthtail(suc(n), s) \equiv nthtail(n, t) \quad \Leftarrow \quad ht(s) \equiv (x, t)$
	$isnil(s) \quad \Leftarrow \quad ht(s) \equiv ()$
	$exists(g, s) \quad \Leftarrow \quad ht(s) \equiv (x, t) \wedge g(x) \equiv true$
	$exists(g, s) \quad \Leftarrow \quad ht(s) \equiv (x, t) \wedge exists(g, t)$
`co-Horn axioms`	$forall(g, s) \quad \Rightarrow \quad (ht(s) \equiv (x, t) \;\Rightarrow\; (g(x) \equiv true \wedge forall(g, t)))$
	$forallExists(g, s) \quad \Rightarrow \quad exists(g, s)$
	$forallExists(g, s) \quad \Rightarrow \quad (ht(s) \equiv (x, t) \;\Rightarrow\; forallExists(g, t))$
(A)	$fair(g, s) \quad \Rightarrow \quad forallExists(g, s)$
(B)	$fair(g, s) \quad \Rightarrow \quad (ht(s) \equiv (x, t) \;\Rightarrow\; (exists(g, s) \wedge fair(g, t)))$
(C)	$fair(g, s) \quad \Rightarrow \quad \exists n, s' : (forall(not \circ g, firstn(n, s)) \wedge nthtail(n, s) \equiv s'$ $\wedge\; ht(s') \equiv (x, t) \wedge g(x) \equiv true \wedge fair(g, t))$
	$infinite(s) \quad \Rightarrow \quad \exists x, t : (ht(s) \equiv (x, t) \wedge infinite(t))$

COLIST serves as the visible subspecification of the following specification of colist comprehension ($filter$). Axioms for $filter$ cannot be included into COLIST because they involve a ν-predicate ($forall$), which must first be translated into a μ-predicate. This happens automatically as a consequence of the step from the swinging type COLIST to the swinging type FILTER where the first one is turned into its *basic Horn translation* (cf.

[78], Def. 2.8).

FILTER = COLIST `then`

`defuncts`	$filter : (entry \rightarrow bool) \times colist \rightarrow colist$
`vars`	$x : entry \quad s : colist \quad g : entry \rightarrow bool$
`Horn axioms`	$filter(g, s) \equiv nil \quad \Leftarrow \quad forall(not \circ g, s)$
	$filter(g, s) \equiv x\&filter(g, t) \quad \Leftarrow \quad ht(s) \equiv (x, t) \wedge g(x) \equiv true$
	$filter(g, s) \equiv filter(g, t) \quad \Leftarrow \quad ht(s) \equiv (x, t) \wedge g(x) \equiv false \wedge exists(g, t)$

Two further specifications of $filter$ are presented in [79]: as a destructor defined in terms of *assertions* and as a *cofunction*, i.e. defined in terms of a *coinductive axiomatization*.

4.3 Alternating bit protocol

The alternating bit protocol of [9] is presented as a network of three list processing functions *send*, *sendAck* and *receive* and two stream processing functions $ch1$ and $ch2$. The elements of a message list xL are equipped with a Boolean tag, transmitted by *send* to the channel $ch1$ and consumed by *receive*. The tags are sent back via the channel $ch2$ to the sender by *sendAck* for confirming (acknowledging) the receipt. Since both tagged messages and acknowledgements may get lost while being transmitted, a message is re-sent until the sender receives the corresponding acknowledgement.

EBPAIR = ENTRY($entry$) `then`

`static preds`	$\_ \not\equiv \_ : (entry \times bool) \times (entry \times bool)$
`vars`	$x, y : entry \quad b, c : bool$
`Horn axioms`	$(x, b) \not\equiv (y, c) \quad \Leftarrow \quad x \not\equiv y$
	$(x, b) \not\equiv (y, c) \quad \Leftarrow \quad b \not\equiv c$

ABP = LIST `and` EBPAIR `and` LIST `and` STREAM `then`

`defuncts`	$send : list(entry) \times bool \times list(bool) \rightarrow list(entry \times bool)$
	$ch1 : list(entry \times bool) \times stream(bool) \rightarrow list(entry \times bool)$
	$receive : list(entry \times bool) \times bool \rightarrow list(entry)$
	$sendAck : list(entry \times bool) \times bool \rightarrow list(bool)$
	$ch2 : list(bool) \times stream(bool) \rightarrow list(bool)$
`static preds`	$net : list(entry) \times bool \times list(entry) \times bool$
`vars`	$x, y : entry \quad xL, yL : list(entry) \quad tag, ack, b : bool \quad acks : list(bool)$
	$pair : entry \times bool \quad pairs : list(entry \times bool) \quad s, s' : stream(bool)$
`Horn axioms`	$send(xL, tag, nil) \equiv nil$
	$send(x : xL, tag, tag : acks) \equiv (x, tag) : send(xL, not(tag), acks)$
	$send(x : xL, tag, ack : acks) \equiv (x, tag) : send(x : xL, tag, acks) \quad \Leftarrow \quad tag \not\equiv ack$
	$ch1(nil, s) \equiv nil$
	$ch1(pair : pairs, s) \equiv pair : ch1(pairs, s') \quad \Leftarrow \quad s \xrightarrow{true} s'$
	$ch1(pair : pairs, s) \equiv ch1(pairs, s') \quad \Leftarrow \quad s \xrightarrow{false} s'$
	$receive(nil, tag) \equiv nil$
	$receive((x, tag) : pairs, tag) \equiv x : receive(pairs, not(tag))$
	$receive((x, tag) : pairs, tag') \equiv receive(pairs, tag') \quad \Leftarrow \quad tag \not\equiv tag'$
	$sendAck(nil, tag) \equiv nil$
	$sendAck((x, tag) : pairs, tag) \equiv tag : sendAck(pairs, not(tag))$
	$sendAck((x, tag) : pairs, tag') \equiv tag : sendAck(pairs, tag') \quad \Leftarrow \quad tag \not\equiv tag'$

$$
\begin{aligned}
&ch2(nil, s) \equiv nil\\
&ch2(ack : acks, s) \equiv ack : ch2(acks, s') \quad \Leftarrow \quad s \stackrel{true}{\longrightarrow} s'\\
&ch2(ack : acks, s) \equiv ch2(acks, s') \quad \Leftarrow \quad s \stackrel{false}{\longrightarrow} s'\\
&net(xL, tag, yL, tag') \quad \Leftarrow \quad send(xL, tag, acks') \equiv pairs \wedge ch1(pairs, s_1) \equiv pairs' \wedge\\
&\qquad receive(pairs', tag') \equiv yL \wedge sendAck(pairs', tag') \equiv acks \wedge\\
&\qquad ch2(acks, s_2) \equiv acks'
\end{aligned}
$$

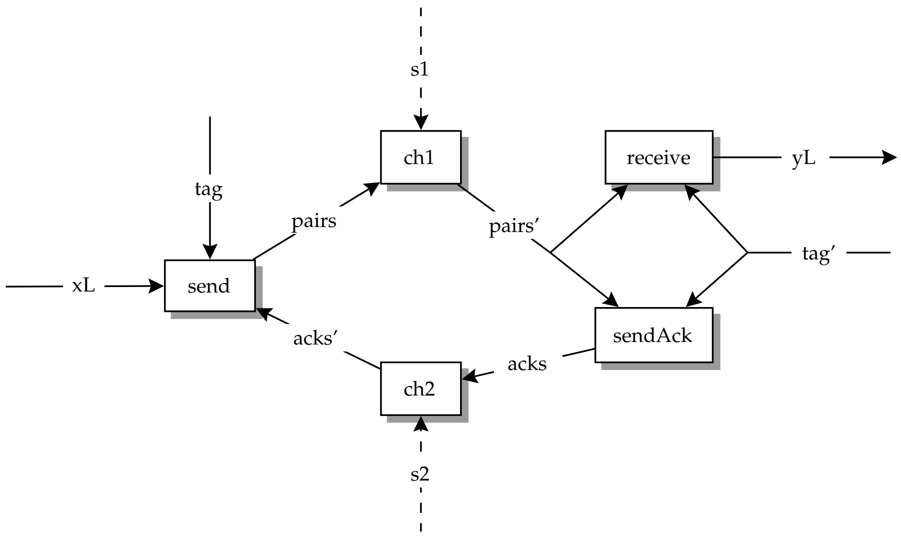

Figure 4. *The functions of ABP.*

The edge labels in Fig. 4 correspond to the arguments of *net*. If each message gets lost at most finitely many times, the list consumed by the sender agrees with the list produced by the receiver, formally,

$$fair(\lambda b.eq(b, true), s_1) \wedge fair(\lambda b.eq(b, true), s_2) \quad \Rightarrow \quad net(xL, tag, xL, tag). \tag{1}$$

We claim that (1) is an inductive theorem of ABP and can be proved similarly to the corresponding conjecture of [72], Section 8.7.

Next we present a completely different specification of the alternating bit protocol that handles also infinite message streams and is designed along the lines of ACCOUNT and COM. All messages have the form a/b. A dynamic atom $s \stackrel{a/b}{\longrightarrow} t$ indicates that the transition from s to t consumes a and produces b.

ABP2 = STREAM **then**

`hidsorts`	*mes sender reveiver channel abp*
`constructs`	$\_/\_ : entry \times entry \to mes$
	$\_/(\_,\_) : bool \times entry \times bool \to mes$
	$(\_,\_)/\_ : entry \times bool \times bool \to mes$
	$receive : bool \times sender \to sender$
	$receive : entry \times bool \times receiver \to receiver$
	$\_\vert\_\vert\_\vert\_ : sender \times channel \times receiver \times channel \to abp$
`destructs`	$str : sender \to stream(entry)$
	$tag : sender \to bool$
	$str : receiver \to stream(entry)$
	$tag : receiver \to bool$
`transpreds`	$\_ \stackrel{\_}{\longrightarrow} \_ : sender \times mes \times sender$
	$\_ \stackrel{\_}{\longrightarrow} \_ : receiver \times mes \times receiver$

	$\_ \stackrel{\_}{\longrightarrow} \_ : channel \times mes \times channel$
	$\_ \longrightarrow \_ : abp \times abp$
`dynamic preds`	$\_ \stackrel{*}{\longrightarrow} \_ : abp \times abp$
	$\Diamond transmits : channel$
ν-`preds`	$fair : channel$
`vars`	$x, y : entry \;\; b, b', c, c' : bool \;\; se, se' : sender \;\; re, re' : receiver \;\; ch, ch', dh, dh' : channel$
	$S, S', S'' : abp \;\; \alpha : mes$
`Horn axioms`	$tag(receive(b, se)) \equiv not(b) \;\Leftarrow\; b \equiv tag(se)$
	$tag(receive(b, se)) \equiv tag(se) \;\Leftarrow\; b \not\equiv tag(se)$
	$str(receive(b, se)) \equiv tail(str(se)) \;\Leftarrow\; b \equiv tag(se)$
	$str(receive(b, se)) \equiv str(se) \;\Leftarrow\; b \not\equiv tag(se)$
	$se \stackrel{b/(x,c)}{\longrightarrow} receive(b, se) \;\Leftarrow\; head(str(se)) \equiv x \wedge tag(se) \equiv c$
	$tag(receive(x, b, re)) \equiv not(t) \;\Leftarrow\; b \equiv tag(re)$
	$tag(receive(x, b, re)) \equiv tag(re) \;\Leftarrow\; b \not\equiv tag(re)$
	$str(receive(x, b, re)) \equiv x\&str(re) \;\Leftarrow\; b \equiv tag(re)$
	$str(receive(x, b, re)) \equiv str(re) \;\Leftarrow\; b \not\equiv tag(re)$
	$re \stackrel{(x,b)/c}{\longrightarrow} receive(x, b, re) \;\Leftarrow\; tag(re) \equiv c$
	$ch \stackrel{x/x}{\longrightarrow} ch$
	$ch \stackrel{\alpha}{\longrightarrow} ch$
	$se\|ch\|re\|dh \longrightarrow se'\|ch'\|re\|dh \;\Leftarrow\; se \stackrel{b/(c,x)}{\longrightarrow} se' \wedge ch \stackrel{(c,x)/(c',y)}{\longrightarrow} ch'$
	$se\|ch\|re\|dh \longrightarrow se\|ch'\|re'\|dh \;\Leftarrow\; ch \stackrel{(b,x)/(b',y)}{\longrightarrow} ch' \wedge re \stackrel{(b',y)/c}{\longrightarrow} re'$
	$se\|ch\|re\|dh \longrightarrow se\|ch\|re'\|dh' \;\Leftarrow\; re \stackrel{(b,x)/c}{\longrightarrow} re' \wedge dh \stackrel{c/c'}{\longrightarrow} dh'$
	$se\|ch\|re\|dh \longrightarrow se'\|ch\|re\|dh' \;\Leftarrow\; dh \stackrel{b/b'}{\longrightarrow} dh' \wedge se \stackrel{b'/(c,x)}{\longrightarrow} se'$
	$\Diamond transmits(ch) \;\Leftarrow\; ch \stackrel{x/x}{\longrightarrow} ch'$
	$\Diamond transmits(ch) \;\Leftarrow\; ch \stackrel{\alpha}{\longrightarrow} ch' \wedge \Diamond transmits(ch')$
	$S \stackrel{*}{\longrightarrow} S$
	$S \stackrel{*}{\longrightarrow} S'' \;\Leftarrow\; S \longrightarrow S' \wedge S' \stackrel{*}{\longrightarrow} S''$
`co-Horn axioms`	$fair(ch) \;\Rightarrow\; \Diamond transmits(ch)$
	$fair(ch) \;\Rightarrow\; (ch \stackrel{\alpha}{\longrightarrow} ch' \;\Rightarrow\; fair(ch'))$

The correctness requirement now reads formally as follows (cf. (1)):

$$fair(ch) \wedge fair(dh) \;\Rightarrow\; \exists\, se', ch', re', dh' : \;\; (se|ch|re|dh \stackrel{*}{\longrightarrow} se'|ch'|re'|dh' \wedge \\ firstn(n, str(se)) \equiv firstn(n, str(re'))).$$

4.4 Processes

STREAM (Section 4.2) specifies linear computation sequences. PROCESS adds nondeterminism via the summation operator $+$. The axioms for the process transition predicate $\longrightarrow: proc \times act \times proc$ are derived from Milner's process calculus CCS ([67], Section 2.5). We include value passing via shared variables (channels). For simplicity, the domain of values is restricted to integers. Valuations are not stored in states as in ACCOUNT and COM. Instead, operators that substitute expressions for variables are included into PROCESS.

The labels of transition systems for process calculi are usually called actions. Here the actions are *silent* (τ) or read ($c?x$), write ($c!e$) commands for which we adopt the notation of Hoare's process language CSP [44]. A specification EXP may provide expression constructors and a substitution operator $\_[\_/\_] : exp \times exp \times var \to exp$. As in COM (cf. Section 3.4), var is supposed to be a subsort of exp.

ACTION(*channel*) = EXP `then`

`sorts`	$channel \quad act$
`constructs`	$\_?\_ : channel \times exp \to act$
	$\_!\_ : channel \times exp \to act$
	$\tau :\to act$
`defuncts`	$\overline{\_} : act \to act$
`Horn axioms`	$\overline{c?e} \equiv c!e$
	$\overline{c!e} \equiv c?e$

PROCESS = ACTION(*channel*) `and` SET `then`

`hidsorts`	$proc$
`constructs`	$stop :\to proc$
	$\_.\_ : act \times proc \to proc$
	$\_ + \_ : proc \times proc \to proc$
	$\_\vert\_ : proc \times proc \to proc$
	$\_ \setminus \_ : proc \times set(act) \to proc$
	$map : (act \to act) \times proc \to proc$
`destructs`	$\_[\_/\_] : proc \times exp \times var \to proc$
`transpreds`	$\_ \stackrel{\_}{\longrightarrow} \_ : proc \times act \times proc$
`vars`	$a : act \quad as : set(act) \quad p, p', q, q' : proc \quad f : act \to act \quad c : channel \quad x, y : var$
	$e, e' : exp$
`Horn axioms`	$c?x.p \stackrel{c?e}{\longrightarrow} p[e/x]$
	$c!e.p \stackrel{c?e}{\longrightarrow} p$
	$\tau.p \stackrel{a}{\longrightarrow} q \quad \Leftarrow \quad p \stackrel{a}{\longrightarrow} q$
	$b.\tau.p \stackrel{a}{\longrightarrow} q \quad \Leftarrow \quad b.p \stackrel{a}{\longrightarrow} q$
	$p + p' \stackrel{a}{\longrightarrow} q \quad \Leftarrow \quad p \stackrel{a}{\longrightarrow} q$
	$p + p' \stackrel{a}{\longrightarrow} q \quad \Leftarrow \quad p' \stackrel{a}{\longrightarrow} q$
	$p\vert p' \stackrel{a}{\longrightarrow} q\vert p' \quad \Leftarrow \quad p \stackrel{a}{\longrightarrow} q$
	$p\vert p' \stackrel{a}{\longrightarrow} p\vert q' \quad \Leftarrow \quad p' \stackrel{a}{\longrightarrow} q'$
	$p\vert p' \stackrel{\tau}{\longrightarrow} q\vert q' \quad \Leftarrow \quad p \stackrel{a}{\longrightarrow} q \wedge p' \stackrel{\overline{a}}{\longrightarrow} q'$
	$p \setminus as \stackrel{a}{\longrightarrow} q \setminus as \quad \Leftarrow \quad p \stackrel{a}{\longrightarrow} q \wedge a \not\in as$
	$map(f, p) \stackrel{f(a)}{\longrightarrow} map(f, q) \quad \Leftarrow \quad p \stackrel{a}{\longrightarrow} q$
	$stop[e/x] \equiv stop$
	$(c!e.p)[e'/x] \equiv c!e[e'/x].p[e'/x]$
	$(c?x.p)[e/x] \equiv c?x.p$
	$(c?x.p)[e/y] \equiv c?x.p[e/y] \quad \Leftarrow \quad x \not\equiv y$
	$(p + p')[e/x] \equiv p[e/x] + p'[e/x]$

Further process constructors may be specified by "head normal form" or "guarded" process equations such as $p \equiv \sum_{i=1}^{n} a_i.p_i$, which stands for the n axioms $p \stackrel{a_1}{\longrightarrow} p_1, \ldots, p \stackrel{a_n}{\longrightarrow} p_n$. For instance, the process equation $clock \equiv tick.clock$ corresponds to the axiom $clock \stackrel{tick}{\longrightarrow} clock$ for the constructors $tick :\to act$ and $clock :\to proc$. By adding this to PROCESS one obtains $clock \sim tick.clock$ as an the inductive theorem of PROCESS. It is easy to see that a process must be guarded for being specifiable in terms of $\longrightarrow$.

Exercises. (A) Show that the τ-*laws*:

$$a.\tau.p \quad \sim \quad a.p \tag{1}$$

$$p + \tau.p \quad \sim \quad \tau.p \tag{2}$$

$$a.(p + \tau.q) + a.q \quad \sim \quad a.(p + \tau.q) \tag{3}$$

are inductive theorems of PROCESS (cf. [67], Prop. 3.2).

(B) Show that the *expansion law* for two guarded processes:

$$p \equiv \sum_{i=1}^{n} a_i.p_i \ \wedge \ q \equiv \sum_{i=1}^{n} b_i.q_i \quad \Rightarrow \quad (p|q) \setminus c \ \sim \begin{cases} \sum_{i=1}^{n} a_i.(p_i|q) \setminus c + \\ \sum_{i=1}^{n} b_i.(p|q_i) \setminus c \ + \\ \sum_{i<j:a_i=d?x,b_j=d!e} \tau.(p_i[e/x]|q_j) \setminus c \ + \\ \sum_{i<j:a_i=d!e,b_j=d?x} \tau.(p_i|q_j[e/x]) \setminus c \end{cases} \tag{4}$$

is an inductive theorem of PROCESS (cf. [67], Cor. 3.6). Use coinduction!

(C) Given k process expressions (= *proc*-normal forms) $\sum_{j=1}^{n_1} a_{1j}.p_{1j}, \ldots, \sum_{j=1}^{n_k} a_{kj}.p_{kj}$ containing occurrences of k guarded process variables (= *proc*-constants) $P_1, \ldots, P_k$[5], show that, for distinct process variables $Q_1, \ldots, Q_k$, the *unique-solution law*:

$$\begin{array}{rl} & P_1 \equiv \sum_{j=1}^{n_1} a_{1j}.p_{1j} \ \wedge \ \ldots \ \wedge \\ & P_k \equiv \sum_{j=1}^{n_k} a_{kj}.p_{kj} \ \wedge \\ & Q_1 \equiv \sum_{j=1}^{n_1} a_{1j}.p_{1j}[Q_1/P_1, \ldots, Q_k/P_k] \ \wedge \ \ldots \ \wedge \\ & Q_k \equiv \sum_{j=1}^{n_k} a_{kj}.p_{kj}[Q_1/P_1, \ldots, Q_k/P_k] \\ \Rightarrow & P_1 \sim Q_1 \ \wedge \ \ldots \ \wedge \ P_k \sim Q_k \end{array} \tag{5}$$

is an inductive theorem of PROCESS (cf. [67], Prop. 3.4).

(D) The following buffer processes are taken from [67], Ex. 3.2.

BUFFERS = PROCESS **then**

`constructs`	$buf_1, buf_2, buf_3 :\to proc$
	$\alpha, \beta, \gamma :\to channel$
`Horn axioms`	$buf_1 \equiv \alpha?.\gamma!.buf_1$
	$buf_2 \equiv \gamma?.\beta!.buf_2$
	$buf_3 \equiv (\alpha?.\beta!.buf_3) + (\beta!.\alpha?.buf_3)$

Show that

$$(buf_1|buf_2) \setminus \gamma? \ \sim \ \alpha?.buf_3 \tag{6}$$

is an inductive theorem of PROCESS. Use (1), (4) and (5) (cf. [67], Ex. 3.2).

4.5 The π-calculus

The π-calculus [68] is a further development of CCS that captures, besides concurrency and communication, the mobility of systems, mainly by treating channels as values. The syntax splits into *names* $(x, y, z, \ldots)$, *labels*, *processes* and *agents* ([68], §9.1):

LABEL(*name*)

`sorts`	*name* *label*
`constructs`	$\_ : name \to label$
	$\bar{\_} : name \to label$

PROCESS = LABEL(*name*) **and** SET **then**

`hidsorts`	*proc* *agent*

[5] P is *guarded* if $P = p_{ij}$ implies $a_{ij} \neq \tau$.

`constructs`	$0 :\to proc$	
	$\tau.\_ : proc \to proc$	silent action
	$(\_).\_ : list(name) \times proc \to agent$	receive list
	$\nu\_\langle\_\rangle.\_ : set(name) \times list(name) \times proc \to agent$	send list
	$\_\ \_ : name \times agent \to proc$	
	$\_ + \_ : proc \times proc \to proc$	summation
	$\_\vert\_ : proc \times proc \to proc$	composition
	$\nu\_\ \_ : set(name) \times proc \to proc$	restriction
	$!\_ : proc \to proc$	replication
`defuncts`	$\_\vert_{agent}\_ : agent \times proc \to agent$	
	$\nu_{agent}\_\ \_ : agent \times proc \to agent$	
`vars`	$x, y : list(name) \quad P, Q : proc \quad xs, ys, zs, xs', zs' : set(name)$	
`Horn axioms`	$((x).P)\vert_{agent}Q \ \equiv\ (x).(P\vert Q)$	
	$(\nu xs\langle y\rangle.P)\vert_{agent}Q \ \equiv\ \nu xs\langle y\rangle.(P\vert Q)$	
	$\nu_{agent}(xs, (y).P) \ \equiv\ (y).\nu xs\ P$	
	$\nu_{agent}(xs, \nu zs\langle y\rangle.P) \ \equiv\ \nu zs'\langle y\rangle.\nu xs'.P$	
	$\Leftarrow \quad y \equiv (y_1, \ldots, y_n) \wedge ys \equiv \{y_1, \ldots, y_n\} \wedge xs' \equiv xs \setminus ys \wedge zs' \equiv zs \cup (xs \cap ys)$	

The **structural congruence** ([68], Def. 9.7) of the μ-calculus is the least equivalence relation that is compatible with the above constructors and the change of bound names (α-conversion) and satisfies the following equations:

$$P + (Q + R) \equiv (P + Q) + R \qquad P + Q \equiv Q + P \qquad P|(Q|R) \equiv (P|Q)|R \qquad P|Q \equiv Q|P$$
$$P|0 \equiv P \qquad !P \equiv P|!P \qquad \nu xs0 \equiv 0$$
$$\nu xs(P|Q) \equiv P|\nu xsQ \quad (\text{if } xs \cap freeVars(P) = \emptyset)$$

Equational axioms that relate process constructors to each other are also typical for *process algebra*. For instance, BPA and ACP (cf. [7]) form equational specifications (mainly of + resp. + and —) and induce a semantics of processes that is defined by the respective initial BPA- resp. ACP-model. The number and complexity of these equations is much greater than what we know from classical algebra with its study of groups, rings, fields, ring modules or vector spaces. The point is that here the initial, free or other standard models have canonical representations, while the quotients given by Ini(BPA) or Ini(ACP) consist of equivalence classes where conceivable unique representatives (normal forms) are difficult to find. This also applies to the above structural congruence of the π-calculus and to the structural congruence of the ambient calculus (cf. [15]).

In general, most functions occurring in equations generating a structural congruence $\equiv$ cannot be declared as defined functions and thus the equations cannot be declared as axioms of a swinging type. Instead, one may search for axioms for a swinging type SP whose behavioral equivalence agrees with the respective structural congruence. Even this often fails, but there remains the possibility to specify a behavioral equivalence that *includes* the structural congruence. This seems to be adequate because in all the above-mentioned cases the structural congruence yields a subrelation of the process equivalence(s) process algebra, the π-calculus or the ambient calculus work with. For instance, after having defined the above structural congruence $\equiv$, Milner presents **reaction rules** (Horn axioms for a dynamic predicate $\longrightarrow$) that include a rule (STRUCT) expressing the compatibility of $\equiv$ with $\longrightarrow$ ([68], Def. 9.16):

$$\begin{array}{rcll} \tau.P + M \longrightarrow P & & & \text{TAU} \\ (x(y).P + M)|(\overline{x}\langle z\rangle.Q + N) \longrightarrow P[z/y]|Q & & & \text{REACT} \\ P|Q \longrightarrow P'|Q & \Leftarrow & P \longrightarrow P' & \text{PAR} \\ \nu xsP|Q \longrightarrow \nu xsP'|Q & \Leftarrow & P \longrightarrow P' & \text{RES} \\ Q \longrightarrow Q' & \Leftarrow & P \longrightarrow P' \wedge P \equiv Q \wedge P' \equiv Q' & \text{STRUCT} \end{array}$$

STRUCT implies that $\equiv$ is a **strong bisimulation** w.r.t. $\longrightarrow$, i.e. zigzag compatible with $\longrightarrow$, in other words: $\equiv$ is a subrelation of behavioral SP-equivalence if $\longrightarrow$ is the only observer of SP. In fact, the question arises whether a structural congruence needs to be separated at all. For instance, what would we lose by replacing the axiom $!P \equiv P|!P$ with an additional reaction rule, namely:

$$!P \longrightarrow Q \quad \Leftarrow \quad P|!P \longrightarrow Q \qquad\qquad \text{REPL}$$

?

The **commitment rules** ([68], Def. 12.6) define a ternary transition relation $\longrightarrow$ analogously to the one specified in Section 4.4. Here $\longrightarrow$ transforms processes into agents. Let α be a label, P, Q, R, S be processes, M, N be summations, A be an agent, $xs \subseteq \{z_1, \dots, z_n\}$ and $z = (z_1, \dots, z_n)$.

$$\begin{array}{rcll}
M + \alpha A + N \xrightarrow{\alpha} A & & & \text{SUM} \\
actP|Q\tau\nu xs(R[z/y]|S) & \Leftarrow & P \xrightarrow{x} (y).R \ \wedge\ Q \xrightarrow{\overline{x}} \nu xs\langle z\rangle.S & \text{L-REACT} \\
P|Q \xrightarrow{\tau} \nu xs(R[z/y]|S) & \Leftarrow & P \xrightarrow{\overline{x}} \nu xs\langle z\rangle.S \ \wedge\ Q \xrightarrow{x} (y).R & \text{R-REACT} \\
P|Q \xrightarrow{\alpha} A|_{agent}Q & \Leftarrow & P \xrightarrow{\alpha} A & \text{L-PAR} \\
P|Q \xrightarrow{\alpha} A|_{agent}P & \Leftarrow & Q \xrightarrow{\alpha} A & \text{R-PAR} \\
\nu xsP \xrightarrow{\alpha} \nu_{agent}(xs, A) & \Leftarrow & P \xrightarrow{\alpha} A \quad \text{if } \alpha \notin \{x, \overline{x}\} & \text{RES} \\
!P \xrightarrow{\alpha} A & \Leftarrow & P|!P \xrightarrow{\alpha} A & \text{REP}
\end{array}$$

Milner shows that the above structural congruence is also zigzag compatible with the transition relation generated by the commitment rules (cf. [68], Thm. 12.8). Hence, again, the above structural congruence $\equiv$ is a subrelation a process congruence, namely the *greatest* strong bisimulation w.r.t. $\longrightarrow$. This is similar to the greatest relation $\sim$ that satisfies the following co-Horn clauses, called **strong equivalence** ([68], Def. 12.13): Let α be a label, P, Q be processes, A, B be agents, $xs \subseteq \{y_1, \dots, y_n\}$, $y = (y_1, \dots, y_n)$ and $z = (z_1, \dots, z_n)$.

$$\begin{array}{rcll}
P \sim Q \ \wedge\ P \xrightarrow{\alpha} A & \Rightarrow & \exists\, B : Q \xrightarrow{\alpha} B \ \wedge\ A \sim B & \\
P \sim Q \ \wedge\ Q \xrightarrow{\alpha} B & \Rightarrow & \exists\, A : P \xrightarrow{\alpha} A \ \wedge\ A \sim B & \\
(y).P \sim (y).Q & \Rightarrow & \forall\, x : P[x/y] \sim Q[x/y] & (7) \\
\nu xs\langle y\rangle.P \sim \nu xs\langle y\rangle.Q & \Rightarrow & P \sim Q & (8)
\end{array}$$

Strong equivalence is not weak enough for obtaining unique solutions of process equations, analogously to (5) above. Hence Milner defines a **weak** or **observation equivalence** ([68], Def. 13.2) for mobile processes similarly to weak bisimilarity for CCS. Weak equivalence has the desired uniqueness property ([68], Thm. 13.8). Both equivalences are *agent congruences* ([68], Def. 12.24), i.e., they are compatible with summation, composition, restriction, replication and, by definition, satisfy the inverses of (7) and (8) ([68], Props. 12.25 and 13.7).

4.6 Infinite trees

In accordance with the structural congruence generated by the equations

$$p + (q + r) \ \equiv\ (p + q) + r, \quad p + q \ \equiv\ q + p, \quad p + p \ \equiv\ p$$

the following specification distinguishes between processes (sort *proc*) and sets of processes (sort *procs*). Since both *proc* and *procs* are hidden sorts, set membership ($\ni$) is declared as a transition predicate, in contrast to SET (cf. Section 2.2.2) where set membership is the Boolean destructor *in*. Otherwise the axioms for some process operators would not be coinductive (cf. [75], Def. 6.1).

FPROCESS = ACT(*act*) `then`

hidsorts $proc \quad procs$

constructs $\_.\_ : act \times procs \to proc$

$\emptyset :\to procs$

$\{\_\} : proc \to procs$

$\_ + \_ : procs \times procs \to procs$

$\_;\_ : procs \times procs \to procs$

$\_|\_ : procs \times procs \to procs$

$map : (act \to act) \times proc \to proc$

$map : (act \to act) \times procs \to procs$

destructs $root : proc \to act$

$subs : proc \to procs$

transpreds $\_ \ni \_ : procs \times proc$

vars $a, b : act \quad p, q : proc \quad ps, ps', qs, qs' : procs \quad f : act \to act$

Horn axioms $root(a.ps) \equiv a$

$subs(a.ps) \equiv ps$

$\{p\} \ni p$

$ps + qs \ni p \quad \Leftarrow \quad ps \ni p$

$ps + qs \ni p \quad \Leftarrow \quad qs \ni p$

$ps; qs \ni a.(ps'; qs) \quad \Leftarrow \quad ps \ni p \wedge root(p) \equiv a \wedge subs(p) \equiv ps'$

$ps|qs \ni a.(ps'|qs) \quad \Leftarrow \quad ps \ni p \wedge root(p) \equiv a \wedge subs(p) \equiv ps'$

$ps|qs \ni a.(ps|qs') \quad \Leftarrow \quad qs \ni q \wedge root(q) \equiv a \wedge subs(q) \equiv qs'$

$ps|qs \ni \tau.(ps'|qs') \quad \Leftarrow \quad ps \ni p \wedge root(p) \equiv a \wedge subs(p) \equiv ps' \wedge$
$\qquad qs \ni q \wedge root(q) \equiv b \wedge subs(q) \equiv qs' \wedge \overline{a} \equiv b$

$root(map(f, p)) \equiv f(root(p))$

$subs(map(f, p)) \equiv map(f, subs(p))$

$map(f, ps) \ni map(f, p) \quad \Leftarrow \quad ps \ni p$

The schema for specifying finite or infinite trees with arbitrary finite outdegree and node labels of sort *entry* generalizes the schema for specifying finite or infinite lists that is used in COLIST (cf. Section 4.2). Concerning the destructors, we follow the schema of REGGRAPH (cf. Section 2.2) and provide a function rs that maps a tree to its root and the tuple of its maximal proper subtrees where the tuple belongs to a sum domain.

FTREE = ENTRY(*entry*) **then**

hidsorts $tree = tree(entry)$

constructs $mt :\to tree$

$\_\&\_ : entry \times tree^n \to tree$ $\qquad n > 0$

destructs $rs : tree \to 1 + (entry \times \amalg_{n>0} tree^n)$

vars $x : entry \quad t_1, \ldots, t_n : tree$

Horn axioms $rs(mt) \equiv ()$

$rs(x\&(t_1, \ldots, t_n)) \equiv (x, t_1, \ldots, t_n)$

Analogously to FTREE, the following specification provides finite and infinite trees with *edge labels* of sort *entry* and finite outdegree.

ETREE = ENTRY(*entry*) **then**

hidsorts $etree = etree(entry)$

constructs $mt :\to etree$

$\_.\_ : entry \times etree \to etree$

$\_ + \_ : etree \times etree \to etree$

`destructs`	$sucs : etree \times entry \to etree^*$	successors
`vars`	$x : entry \quad t, t', t_1, \dots, t_m, t'_1, \dots, t'_n : etree$	
`Horn axioms`	$sucs(mt, x) \equiv ()$	
	$sucs(x.t, x) \equiv (t)$	
	$sucs(x.t, y) \equiv () \quad \Leftarrow \quad x \not\equiv y$	
	$sucs(x, t + t') \equiv (t_1, \dots, t_m, t'_1, \dots, t'_n)$	
	$\quad \Leftarrow \quad sucs(x, t) \equiv (t_1, \dots, t_m) \ \wedge \ sucs(x, t') \equiv (t'_1, \dots, t'_n)$	
	$sucs(x, t + t') \equiv sucs(t) \quad \Leftarrow \quad sucs(x, t') \equiv ()$	
	$sucs(x, t + t') \equiv sucs(t') \quad \Leftarrow \quad sucs(x, t) \equiv ()$	

ETREE can be extended to a specification of processes (cf. Section 4.4) by axiomatizing behavioral process equivalence in terms of a transition predicate that is derived from the *etree*-destructor *sucs*.

EPROCESS = ETREE `then`

`hidsorts`	$eproc = eproc(entry)$	
`constructs`	$mkproc : etree \to eproc$	
`defuncts`	$mt : \to eproc$	
	$\_.\_ : entry \times eproc \to eproc$	
	$\_ + \_ : eproc \times eproc \to eproc$	
`transpreds`	$\_ \stackrel{\_}{\longrightarrow} \_ : eproc \times entry \times eproc$	
`vars`	$x : entry \quad t, t_1, \dots, t_n : etree \quad p, p' : eproc$	
`Horn axioms`	$mkproc(t) \stackrel{x}{\longrightarrow} mkproc(t_i) \quad \Leftarrow \quad sucs(t, x) \equiv (t_1, \dots, t_n)$	for all $1 \leq i \leq n$
	$mt \equiv mkproc(mt)$	
	$x.mkproc(t) \equiv mkproc(x.t)$	
	$mkproc(t) + mkproc(t') \equiv mkproc(t + t')$	

Here the collection of successors of a tree is regarded as a set of trees, not as a list as in ETREE.

5 Petri nets

5.1 Weighted sets

The place domains of high-level Petri nets are finite multisets with positive or negative cardinalities (cf., e.g., [50], Vol. 2, Section 4.2). Hence weighted sets are specified similarly to bags (cf. Section 2.2.3). Let INT be a specification of integer arithmetic such as the one given in Section 2.1.

WSET = LIST `and` INT `then`

`hidsorts`	$wset = wset(entry)$
`constructs`	$empty : \to wset$
	$[\_] : entry \to wset$
	$\_ + \_ : wset \times wset \to wset$
	$-\_ : wset \to wset$
`destructs`	$weight : wset \times entry \to int$
`defuncts`	$mkwset : list \to wset$
	$\_ - \_ : wset \times wset \to wset$
	$\_ * \_ : nat \times entry \to wset$
	$map : (entry \to entry) \times wset \to wset$
	$filter : (entry \to bool) \times wset \to wset$

	$\lvert\_\rvert : wset \to int$
`preds`	$\_ \in \_ : entry \times wset$
`copreds`	$\_ \subseteq \_ : wset \times wset$
`vars`	$x, y : entry \quad V, W : wset \quad n : nat \quad f : entry \to entry \quad g : entry \to bool \quad L, L' : list$
`Horn axioms`	$weight(empty, x) \equiv 0$
	$weight([x], x) \equiv 1$
	$weight([x], y) \equiv 0 \quad \Leftarrow \quad x \not\equiv y$
	$weight(V + W, x) \equiv weight(V, x) + weight(W, x)$
	$weight(-W, x) \equiv -weight(W, x)$
	$mkwset(nil) \equiv empty$
	$mkwset(x : L) \equiv [x] + mkwset(L)$
	$V - W \equiv V + (-W)$
	$0 * x \equiv empty$
	$suc(n) * x \equiv [x] + (n * x)$
	$map(f, empty) \equiv empty$
	$map(f, [x]) \equiv [f(x)]$
	$map(f, V + W) \equiv map(f, V) + map(f, W)$
	$map(f, -W) \equiv -map(f, W)$
	$filter(g, empty) \equiv empty$
	$filter(g, [x]) \equiv [x] \quad \Leftarrow \quad g(x) \equiv true$
	$filter(g, [x]) \equiv empty \quad \Leftarrow \quad g(x) \equiv false$
	$filter(g, V + W) \equiv filter(g, V) + filter(g, W)$
	$filter(g, -W) \equiv -filter(g, W)$
	$\lvert empty\rvert \equiv 0$
	$\lvert [x]\rvert \equiv 1$
	$\lvert V + W\rvert \equiv \lvert V\rvert + \lvert W\rvert$
	$\lvert -W\rvert \equiv -\lvert W\rvert$
	$x \in W \quad \Leftarrow \quad weight(W, x) > 0$
`co-Horn axioms`	$V \subseteq W \quad \Rightarrow \quad (x \in V \ \Rightarrow \ x \in W)$

Note that weighted sets built up with the above constructors yield an Abelian group, i.e., + is associative and commutative, *empty* is neutral (or absorbing) w.r.t. + and $-W$ is the inverse of W w.r.t. +. This fact will be referred to in Section 5.3 where *invariants* are derived from linear functions mapping *states* (= tuples of weighted sets) to powers of an Abelian group.

5.2 Nets

Our definition of nets combines the definitions of *predicate/event* nets, *high-level nets* ([87], Def. 4.2) with *flexible arc inscriptions* ([87], Section 10.4), *colored nets* ([50], Def. 2.5) and *algebraic nets* ([52], Section 2.2). All these approaches associate a domain of structured tokens with each place of a net N. Given that $p_1, \ldots, p_n$ are the places of N, we denote the domain of p_i-tokens by the sort dom_i. Moreover, let $t_1, \ldots, t_k$ be the transitions of N. Each arc connects a place with a transition and is labelled with the pattern of a weighted set of elements taken from resp. added to the place whenever the transition fires. Hence the label of an arc connecting place p_i with transition t_j is a term of sort $wset(dom_i)$. If the arc leads from p_i to t_j, the term is denoted by $in_{i,j}$. If it leads from t_j to p_i, the term is denoted by $out_{j,i}$. The term

$$inout_{i,j} \;=\; out_{j,i} - in_{i,j}$$

describes the effect of t_j on p_i. Global state changes may involve all places and transitions of N and imagined as concurrent movements of objects between the places of N. They are represented by the **incidence matrix**:

$$\begin{pmatrix} inout_{1,1} & \cdots & inout_{1,k} \\ & \cdot & \\ & \cdot & \\ & \cdot & \\ inout_{n,1} & \cdots & inout_{n,k} \end{pmatrix}$$

Moreover, a transition t_j may be associated with a Boolean term $guard_j$ that restrict the instances of $in_{i,j}$ and $out_{j,i}$ to the subsets whose elements satisfy $guard_j$. of

N is usually depicted as a labelled bipartite graph whose nodes are the places resp. transitions of N. A transition is labelled with its guard if there is any. There are edges from p_i to t_j labelled with $in_{i,j}$ and from t_j to p_i labelled with $out_{j,i}$ unless $in_{i,j}$ resp. $out_{j,i}$ map all their arguments to the empty weighted set.

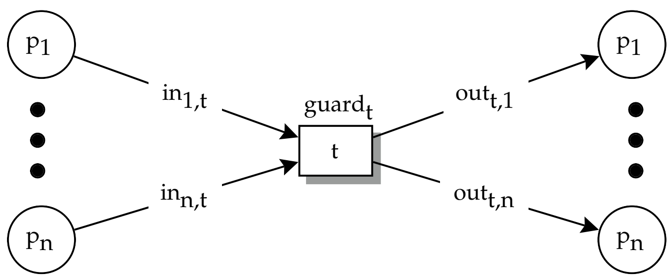

Figure 5. *The functions and predicates associated with a transition.*

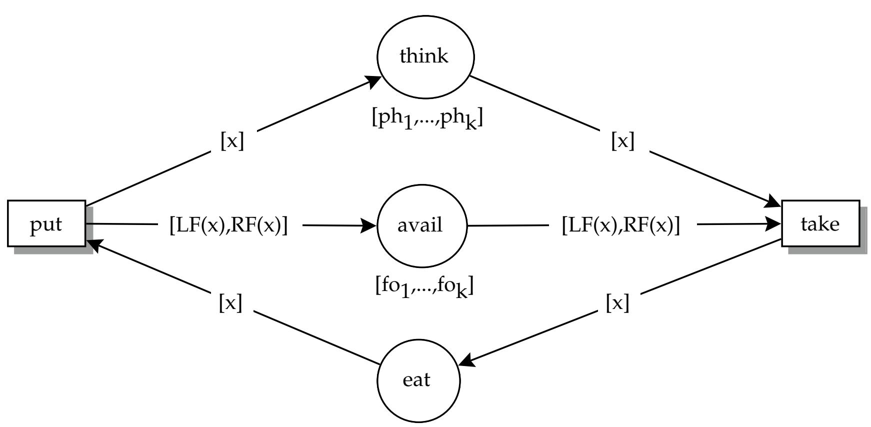

Figure 6. *DPH with initial marking M.*

Example 5.2.1 (dining philosophers) (cf. [87], Section 1; [50], Vol. 2, Section 1.6); [88], Fig. 21.3) The net DPH of Fig. 6 has three places: *think* (thinking philosophers), *eat* (eating philosophers) and *avail* (available forks), and two transitions: *take* (take fork) and *put* (release fork).

$spec(DPH)$ = INT `and` WSET `then`

`sorts`	$dom_{think} = dom_{eat} = phil$	$dom_{avail} = fork$
`constructs`	$ph : int \to phil$	$fo : int \to fork$
`defuncts`	$max : \to int$	$LF, RF : phil \to fork$
`vars`	$i : int \quad x : phil$	
`Horn axioms`	$LF(ph(i)) \equiv fo(i)$	

$$
\begin{array}{ll}
RF(ph(i)) \equiv fo(i+1) \quad \Leftarrow \quad i < max & \\
RF(ph(max)) \equiv fo(1) & \\
out_{put,think} \equiv in_{think,take} \equiv out_{take,eat} \equiv in_{eat,put} \equiv [x] & \\
out_{put,avail} \equiv in_{avail,take} \equiv [LF(x), RF(x)]^{6} & \\
out_{t,p} \equiv empty & \text{for all } (t,p) \notin \{(put,think),(take,eat),(put,avail) \\
in_{p,t} \equiv empty & \text{for all } (p,t) \notin \{(think,take),(avail,take)
\end{array}
$$

Definition 5.2.2 (dynamics of N) $spec(N)$ specifies the (static) data types involved in N. The behavior or dynamics of N can be presented as an extension of $spec(N)$:

$dyn(N) = spec(N)$ `then`

$$
\begin{array}{lll}
\texttt{hidsorts} & state = wset(dom_1) \times \ldots \times wset(dom_n) & \\
\texttt{defuncts} & p_i : state \to wset(dom_i) & 1 \le i \le n \\
\texttt{preds} & enabled_j : state & 1 \le j \le k \\
 & enabled : state & \\
 & \langle . \rangle r,\ \Diamond r : state & \text{for all } r : state \\
 & disjoint_{j,j}, overlap_{j,j}, same_{j,j}, different_{j,j'}, conflict_{j,j'} : state & 1 \le j, j' \le k \\
\texttt{transpreds} & \_ \xrightarrow{t_j} \_ : state \times state & 1 \le j \le k \\
 & \_ \to \_ : state \times state & \\
\texttt{copreds} & disabled : state & \\
 & [.]r,\ \Box r,\ AF(r) : state & \text{for all } r : state \ (AF = \text{``always finally''}) \\
 & q \leadsto r : state & \text{for all } q, r : state \\
\texttt{vars} & s, s' : state \quad ws_1 : wset(dom_1) \ldots ws_n : wset(dom_n) & \\
\texttt{Horn axioms} & p_i(ws_1, \ldots, ws_n) \equiv ws_i & 1 \le i \le n \\
 & enabled_j(s) \quad \Leftarrow \quad guard_j \wedge \bigwedge_{i=1}^{n} in_{i,j} \subseteq p_i(s) & 1 \le j \le k \\
 & s \xrightarrow{t_j} (p_1(s) + inout_{1,j}, \ldots, p_n(s) + inout_{n,j}) \quad \Leftarrow \quad enabled_j(s) & 1 \le j \le k \\
 & s \to s' \quad \Leftarrow \quad s \xrightarrow{t_j} s' & 1 \le j \le k \\
 & enabled(s) \quad \Leftarrow \quad s \to s' & \\
 & \langle . \rangle r(s) \quad \Leftarrow \quad s \to s' \wedge r(s') & \\
 & \Diamond r(s) \quad \Leftarrow \quad r(s) & \\
 & \Diamond r(s) \quad \Leftarrow \quad s \to s' \wedge \Diamond r(s') & \\
 & disjoint_{j,j'}(s) \quad \Leftarrow \quad \forall\, x_1, \ldots, x_n : \bigwedge_{i=1}^{n} in_{i,j} \cap in_{i,j'} = \emptyset & (1) \\
 & overlap_{j,j'}(s) \quad \Leftarrow \quad x \in in_{i,j} \wedge x \in in_{i,j'} & 1 \le i \le n \\
 & same_{j,j'}(s) \quad \Leftarrow \quad \forall\, x_1, \ldots, x_n : \bigwedge_{i=1}^{n} (in_{i,j} \equiv in_{i,j'} \wedge out_{j,i} \equiv out_{j',i}) & (2) \\
 & different_{j,j'}(s) \quad \Leftarrow \quad in_{i,j} \not\equiv in_{i,j'} & 1 \le i \le n \\
 & different_{j,j'}(s) \quad \Leftarrow \quad out_{j,i} \not\equiv out_{j',i} & 1 \le i \le n \\
 & conflict_{j,j'}(s) \quad \Leftarrow \quad enabled_j(s) \wedge enabled_{j'}(s) \wedge & \\
 & \qquad\qquad overlap_{j,j'}(s) \wedge different_{j,j'}(s) & 1 \le i \le n \\
\texttt{co-Horn axioms} & disabled(s) \quad \Rightarrow \quad (s \to s' \Rightarrow False) & \\
 & [.]r(s) \quad \Rightarrow \quad (s \to s' \Rightarrow r(s')) & \\
 & \Box r(s) \quad \Rightarrow \quad r(s) & \\
 & \Box r(s) \quad \Rightarrow \quad (s \to s' \Rightarrow \Box r(s')) & \\
 & AF(r)(s) \quad \Rightarrow \quad (r(s) \ \vee \ \exists\, s' : s \to s') & \\
 & AF(r)(s) \quad \Rightarrow \quad (s \to s' \ \Rightarrow \ (r(s) \ \vee \ AF(r)(s'))) & \\
 & (q \leadsto r)(s) \quad \Rightarrow \quad (q(s) \ \Rightarrow \ AF(r)(s)) & \\
 & (q \leadsto r)(s) \quad \Rightarrow \quad (s \to s' \ \Rightarrow \ (q \leadsto r)(s')) &
\end{array}
$$

[6] $[x_1, \ldots, x_k]$ stands for the weighted set $[x_1] + \cdots + [x_k]$.

A **marking** of (the places of) N is a *state*-sorted ground term over $dyn(N)$. $\mathcal{M}(N)$ denotes the set of markings of N. N **satisfies** a formula φ over $dyn(N)$, written $N \models \varphi$, if the initial model of $dyn(N)$ satisfies φ (cf. [75]). N is **finitely branching** if for all transitions t and markings M of N there are only finitely many markings M' such that $M \stackrel{t}{\longrightarrow} M'$. ❑

Do the axioms for *disjoint*, *same* and *conflict* capture the definitions of these predicates that are given in [103], Section 3.4.1? The variables $x_1, \dots, x_n$ are supposed to include all variables in the premise of axiom (1) resp. (2).

$dyn(N)$ is functional, coinductive and continuous. Hence by [75], Thm. 6.5, behavioral $dyn(N)$-equivalence is a weak congruence and thus by [75], Thm. 3.8(3), $dyn(N)$ enjoys the following Hennessy-Milner Theorem:

Theorem 5.2.3 *Given a net N, two markings of N are behaviorally $dyn(N)$-equivalent iff they satisfy the same poly-modal formulas over $dyn(N)$.* ❑

5.3 Net properties

[103, 22, 53] visualize properties of the modal operator $\leadsto$ (*leads to*) with the help of proof graphs. Inference rules, which build up the derivations presented as poof graphs and which are correct with to the intial model of $dyn(N)$ read as follows:

$\leadsto$-decomposition
$$\frac{\varphi \leadsto \psi}{\varphi \Rightarrow [.]\rho \ \wedge \ \rho \leadsto \psi} \Uparrow \qquad \frac{\varphi \leadsto \psi}{\varphi \Rightarrow \psi} \Uparrow$$
$$\frac{\varphi_1 \ \vee \ \varphi_2 \leadsto \psi}{\varphi_1 \leadsto \psi \ \wedge \ \varphi_2 \leadsto \psi} \Uparrow \qquad \frac{\varphi_1 \ \wedge \ \varphi_2 \leadsto \psi}{\varphi_1 \leadsto \psi \ \vee \ \varphi_2 \leadsto \psi} \Uparrow$$
$$\frac{\exists x : \varphi \leadsto \psi}{\varphi \leadsto \psi} \Uparrow \qquad \frac{\varphi \leadsto \forall x \psi}{\varphi \leadsto \psi} \Uparrow$$

AF-decomposition
$$\frac{AF(\varphi)(M)}{(\lambda s.(s \sim M) \leadsto \varphi)(M)} \Uparrow \quad \text{for all } M \in \mathcal{M}(N)$$

Given an initial marking $M \in \mathcal{M}(N)$, $dyn(N)$-formulas of the form $(\Diamond\varphi)(M)$, $AF(\varphi)(M)$ or $\varphi \leadsto \psi$ are called **reachability conditions**, while those of the form $\Box\varphi(M)$ are called **invariants**. For proving reachability conditions one may use the following instance of fixpoint induction on predicates (cf. [75, 78]): Let t be a term and s, s' be variables.

$\Diamond$-induction
$$\frac{(\Diamond r)(t) \Rightarrow \psi}{r(s) \Rightarrow \psi(s) \quad \wedge \quad (s \to s' \wedge \psi(s')) \Rightarrow \psi(s)} \Updownarrow$$

The conclusion is the instance of the two $\Diamond$-axioms (cf. Def. 5.2.2) with $\Diamond r$ replaced by q. For proving invariants one may use the following instance of coinduction on copredicates (cf. [75, 78]):

$\Box$-coinduction
$$\frac{\psi \Rightarrow (\Box r)(t)}{\psi(s) \Rightarrow r(s) \quad \wedge \quad (s \to s' \wedge \psi(s)) \Rightarrow \psi(s')} \Updownarrow$$

The conclusion is the instance of the two $\Box$-axioms (cf. Def. 5.2.2) with $\Box r$ replaced by q.

The following theorem provides a derived inference rule that employs linear functions on $\mathcal{M}(N)$. It generalizes, for instance, Theorem 4.7 of [50], Vol. 2.

Let A be the initial model of $spec(N)$. Given a sort $dom \in spec(N)$ such that A_{dom} is an Abelian group with addition $\oplus$ and neutral element 0, a tuple $f = (f_1, \dots, f_n)$ of function symbols $f_i : wset(dom_i) \to dom \in spec(N)$ is **linear** if for all $1 \le i \le n$ and weighted sets V, W, N satisfies $f_i(V + W) \equiv f_i(V) \oplus f_i(W)$.

Theorem 5.3.1 (invariant criterion) *Given $M \in \mathcal{M}(N)$, N satisfies $(\Box\varphi)(M)$ if there is a linear function $f = (f_1, \ldots, f_n)$ such that N satisfies*

(1) $\sum_{i=1}^n f_i(p_i(M)) \equiv \sum_{i=1}^n f_i(p_i(s)) \;\Rightarrow\; \varphi(s)$,
(2) $\bigwedge_{j=1}^k (guard_j \;\Rightarrow\; \sum_{i=1}^n f_i(inout_{i,j}) \equiv 0)$.

Proof. Let

$$q(s) \quad =_{def} \quad \sum_{i=1}^n f_i(p_i(M)) \equiv \sum_{i=1}^n f_i(p_i(s)).$$

We show $s \equiv M \Rightarrow \Box q(s)$ by $\Box$-coinduction and conclude $\Box\varphi(M)$ because by (1), $\Box q(s)$ implies $\Box\varphi(s)$. $\Box$-coinduction yields two proof obligations:

(3) N satisfies $q(M)$,
(4) N satisfies $(s \to s' \wedge q(s)) \Rightarrow q(s')$.

(3) holds true trivially. So let $M_1, M_2 \in \mathcal{M}$ such that N satisfies $M_1 \to M_2$ and $q(M_1)$. (4) holds true if $N \models q(M_2)$. $N \models M_1 \to M_2$ implies $N \models M_1 \xrightarrow{j} M_2$ for some transition t_j. Hence N satisfies $p_i(M_1) + out_{j,i} \equiv p_i(M_2) + in_{i,j}$ for all $1 \le i \le n$ and $enabled_j(M_1)$. The latter implies $N \models guard_j$. Since f is linear,

$$f_i(p_i(M_1)) + f_i(out_{j,i}) \equiv f_i(p_i(M_2)) + f_i(in_{i,j}). \tag{5}$$

By (2), N satisfies

$$guard_j(x) \quad \Rightarrow \quad \sum_{i=1}^n f_i(out_{j,i}) \equiv \sum_{i=1}^n f_i(in_{i,j}) \tag{6}$$

(5) implies

$$\sum_{i=1}^n f_i(p_i(M_1)) + \sum_{i=1}^n f_i(out_{j,i}) \equiv \sum_{i=1}^n f_i(p_i(M_2)) + \sum_{i=1}^n f_i(in_{i,j})$$

and thus by (6),

$$\sum_{i=1}^n f_i(p_i(M_1) \equiv \sum_{i=1}^n f_i(p_i(M_2))$$

because N satisfies $guard_j$. Hence $N \models q(M_1)$ implies $N \models q(M_2)$, and the proof of (4) is complete. ❑

Example 5.3.2 (dining philosophers) (cf. Ex. 5.2.1) Starting out from the initial marking of DPH as in Fig. 6, we show three invariants:

- Each philosopher is either thinking or eating.
- Two philosophers eating at the same time do not share forks.
- Potential users of available forks are thinking.

We generalize the last two invariants and come up with the following three conjectures:

(1) Each philosopher is either thinking or eating.
(2) Each fork is either available or in use by a (single) philosopher.
(3) Each fork is either available or its (potential) user does not think.

DPH has three places: *think*, *avail* and *eat*. Hence *dyn*(DPH) has three *state*-destructors that define the markings of DPH such as the initial one (see Fig. 6): $think(M) = [ph_1, \ldots, ph_k]$, $avail(M) = [fo_1, \ldots, fo_k]$ and $eat(M) = empty$. We express (1)-(3) in terms of *dyn*(DPH):

$$\begin{aligned}
\varphi_1 \quad &=_{def} \quad \Box([ph_1, \ldots, ph_k] \equiv think(s) + eat(s))(M),\text{[7]} \\
\varphi_2 \quad &=_{def} \quad \Box([fo_1, \ldots, fo_k] \equiv avail(s) + map(LF)(eat(s)) + map(RF)(eat(s)))(M), \\
\varphi_3 \quad &=_{def} \quad \Box([fo_1, \ldots, fo_k] \equiv avail(s) - map(LF)(think(s)) - map(RF)(think(s)))(M).
\end{aligned}$$

For satisfying Condition 5.3.1(1) we need linear functions $f = (f_1, f_2, f_3) : \mathcal{M}(DPH) \to wset(phil, fork)^3$ such that DPH satisfies $q_f(s) \Rightarrow \varphi_1$, $q_f(s) \Rightarrow \varphi_2$ and $q_f(s) \Rightarrow \varphi_3$, respectively, where $q_f(s)$ is given by the equation

$$f_1(think(s)) + f_2(avail(s)) + f_3(think(s)) \;\equiv\; f_1([ph_1, \dots, ph_k]) + f_2([fo_1, \dots, fo_k]) + f_3(empty).$$

Since all guards of DPH are *True*, the formula 5.3.1(2) amounts to:

$$f_1(-[x]) + f_2(-[LF(x), RF(x)]) + f_3([x]) \equiv empty \;\wedge\; f_1([x]) + f_2([LF(x), RF(x)]) + f_3(-[x]) \equiv empty \quad (4)$$

(cf. Ex. 5.2.1). We obtain

$$\begin{array}{lllll} DPH & \models & \varphi_1 \wedge (4) & \text{for} & f(U,V,W) =_{def} (U, empty, W), \\ DPH & \models & \varphi_2 \wedge (4) & \text{for} & f(U,V,W) =_{def} (empty, V, map(LF)(W) + map(RF)(W)), \\ DPH & \models & \varphi_3 \wedge (4) & \text{for} & f(U,V,W) =_{def} (-map(LF)(W) - map(RF)(W), V, empty). \end{array}$$

Since f is linear in all three cases, we conclude (1)-(3) from Thm. 5.3.1. ❏

Thm. 5.3.1 provides us with a rule for proving net invariants:

invariant rule

$$\frac{\Box\varphi(s)}{\begin{array}{l}\sum_{i=1}^n f_i(p_i(s)) \equiv \sum_{i=1}^n f_i(p_i(s')) \;\;\Rightarrow\;\; \varphi(s') \\ \wedge\; \bigwedge_{j=1}^k (guard_j \;\Rightarrow\; \sum_{i=1}^n f_i(inout_{i,j}) \equiv 0)\end{array}} \Uparrow$$

if for all $1 \le i \le n$, the initial model of $spec(N)$ interprets dom as an Abelian group and $f_i : wset(dom_i) \to dom$ as a linear function

Since for all φ, N satisfies

$$\begin{array}{lll} [.]\varphi(s) & \iff & \forall s' : (s \to s' \Rightarrow \varphi(s')), \\ s \to s' & \Longrightarrow & \bigvee_{j=1}^k p_i(s') \equiv p_i(s) + inout_{i,j}(x), \end{array}$$

the following rule is correct for all $1 \le i \le n$:

step rule

$$\frac{([.] \bigvee_{j=1}^k p_i(s') \equiv p_i(s) + inout_{i,j})(s)}{True} \Updownarrow$$

A net N **terminates in** $M \in \mathcal{M}(N)$ if N satisfies $AF(disabled)(M)$ or, equivalently, for all $1 \le j \le k$, N satisfies $AF(\neg enabled_j(s))(M)$. Sometimes $\varphi \leadsto \psi$ can be decomposed into proofs that (1) all runs starting out from a state satisfying φ terminate and (2) final states satisfy ψ. This amounts to the following expansion rule:

termination rules

$$\frac{\varphi \leadsto \psi}{\varphi \leadsto disabled \;\wedge\; disabled \Rightarrow \psi} \Uparrow$$

$$\frac{AF(disabled)(s)}{\bigwedge_{j=1}^k guard_j \;\Rightarrow\; \sum_{i=1}^n f_i(inout_{i,j}) < 0} \Uparrow$$

if for all $1 \le i \le n$, $Her(spec(N))$ interprets dom as an Abelian group and $f_i : wset(dom_i) \to dom$ as a linear function

Theorem 5.3.3 (termination criterion) *Given $M \in \mathcal{M}(N)$, N satisfies $AF(disabled)(M)$ if there is a linear function $f = (f_1, \dots, f_n)$ such that for all $a, b, c \in A_{dom}$, $a > b$ implies $a + c > b + c$, and f* **decreases**, *i.e., for all $1 \le j \le k$, N satisfies*

$$guard_j \;\;\Rightarrow\;\; \sum_{i=1}^n f_i(inout_{i,j}) < 0. \quad (1)$$

[7]More precisely, $\varphi_1 = (\Box q)(M)$ for some implicit predicate $q : state$ defined by $q(s) = ([ph_1, \dots, ph_k] \equiv think(s) + eat(s))$.

Proof. By (1), N satisfies

$$guard_j \quad \Rightarrow \quad \sum_{i=1}^{n} f_i(out_{j,i}) < \sum_{i=1}^{n} f_i(in_{i,j}(x)). \tag{2}$$

As in the proof of Thm. 5.3.1, $N \models M_1 \to M_2$ implies that N satisfies $guard_j$ and

$$\sum_{i=1}^{n} f_i(p_i(M_1) + \sum_{i=1}^{n} f_i(out_{j,i}) \equiv \sum_{i=1}^{n} f_i(p_i(M_2)) + \sum_{i=1}^{n} f_i(in_{i,j})$$

for some $1 \leq j \leq k$ and a. Hence by (2),

$$\sum_{i=1}^{n} f_i(p_i(M_1)) + \sum_{i=1}^{n} f_i(out_{j,i}) > \sum_{i=1}^{n} f_i(p_i(M_2)) + \sum_{i=1}^{n} f_i(out_{j,i}). \tag{3}$$

(3) implies

$$\sum_{i=1}^{n} f_i(p_i(M_1)) > \sum_{i=1}^{n} f_i(p_i(M_2)). \tag{4}$$

Therefore, $N \models M_1 \stackrel{*}{\longrightarrow} M_2$ also implies (4), and we conclude $N \models AF(disabled)(M)$ because $>$ is well-founded. ❏

The similarity of the proofs of Thms. 5.3.1 and 5.3.3 suggests a generalization that subsumes both results. For this purpose [54] have introduced the notion of a *net simulation* based on *preordered commutative monoids*.

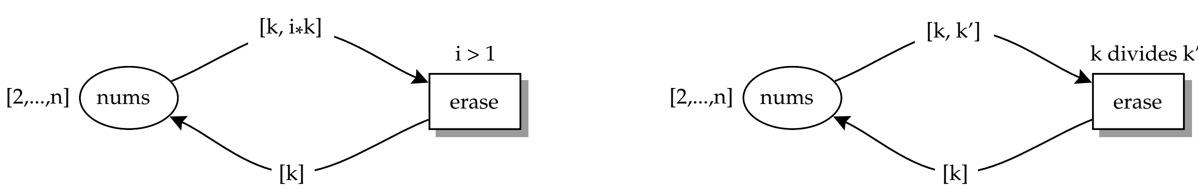

Figure 7. *SIEVE1 and SIEVE2.*

Example 5.3.4 (sieve of Eratosthenes) ([88], Section 15.3) Runs of the nets SIEVE1 and SIEVE2 select all primes among the set of all natural numbers k with $2 \leq k \leq n$. We claim that both nets satisfy the reachability condition

$$nums(s) \equiv [2, \ldots, n] \rightsquigarrow nums(s) \equiv filter(prime)[2, \ldots, n]. \quad ❏$$

The schema of SIEVE2 can be used for specifying many algorithms that amount to set modifications such as, e.g., an algorithm for computing the **shortest paths** in a labelled directed graph $G_0 \subseteq Nodes \times \mathbb{N} \times Nodes$. Without comment we present the corresponding net of [88], Fig. 23.4, in Fig. 8.

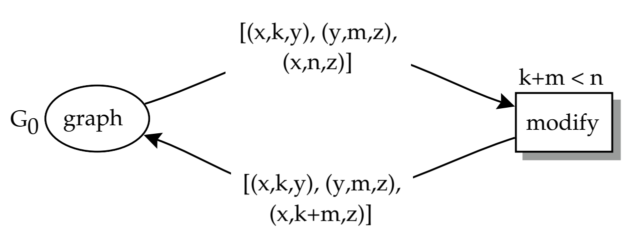

Figure 8. *A net for computing shortest paths.*

Exercise. Specify the net of Fig. 8. Show the correctness of the specified shortest-path algorithm by proving the inductive theorem $r(s) \Rightarrow \Diamond q(s)$ where the axioms for r and q read as follows:

$$r(G_0)$$
$$q(s) \quad \Leftarrow \quad disabled(s) \ \wedge \ forall(\lambda(x,k,y).eq(k, mindist(G_0,x,y)), graph(s))$$

Example 5.3.5 (minimal distances) Runs of the following net MD compute the minimal distances of *inner* nodes from some *root* node of a directed graph $G_0 \subseteq Nodes \times Nodes$.

GRAPH = WSET `then`

`sorts`	$node$
`constructs`	$v_1,\dots,v_k :\to node$
`defuncts`	$G_0 :\to wset(node \times node)$
	$roots, inner :\to wset(node)$
	$sucs : node \to wset(node)$
	$next : nat \times node \to wset(node \times nat \times node)$
`static preds`	$path : wset(node \times node) \times node \times nat \times node$
	$rev, minrev : node \times nat \times node$
	$acyclic : wset(node \times node)$
`vars`	$x, y : node \quad f : node \to node \times nat \quad m, n : nat \quad G : wset(node \times node)$
`Horn axioms`	$G_0 \equiv [\dots]$
	$roots \equiv [\dots]$
	$inner \equiv [v_1, ..., v_k] - roots$
	$weight(sucs(x), y) \equiv weight(G_0, (x, y))$
	$next(n, x) \equiv map(\lambda z.(z, n, x))(sucs(x))$
	$path(G, x, 0, x)$
	$path(G, x, n+1, z) \quad \Leftarrow \quad weight(G, (x, y)) > 0 \wedge path(G, y, n, z)$
	$rev(y, n, x) \quad \Leftarrow \quad y \in sucs(x) \wedge r \in roots \wedge path(G_0, r, n, y)$
`co-Horn axioms`	$minrev(y, m, x) \quad \Rightarrow \quad (rev(y, n, x) \quad \Rightarrow \quad m \le n)$
	$acyclic(G) \quad \Rightarrow \quad (path(G, x, n, x) \quad \Rightarrow \quad n \equiv 0)$

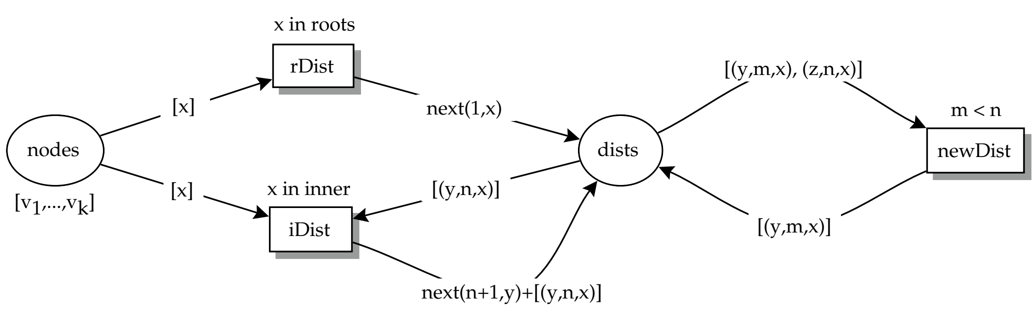

Figure 9. *MD*

Initially, MD has all nodes of the given graph at place *nodes*. Place *dists* stores triples (y, n, x) such that (x, y) is an arc of the graph and n is the length of a path from a root to y. We claim that MD satisfies the reachability condition

$$nodes(s) \equiv [v_1, \dots, v_k] \ \rightsquigarrow \quad nodes(s) \equiv empty \ \wedge$$
$$dists(s) \equiv [(y, n, x) \mid minrev(y, n, x)]^8 \ \wedge$$
$$acyclic(map(\lambda(y, n, x).(x, y))(dists(s))).$$

MD has been inspired by quite similar nets investigated in [52, 53]. ❏

Example 5.3.6 (alternating bit protocol) (cf. Section 4.3) The unsafe transmission of messages and acknowledgements is simulated with the help of Boolean tags taken from places 4 and 7 of ABP (see Fig. 10). Again, the functions occurring in arc inscriptions are part of an underlying domain specification:

DOMAINS = LIST `and` WSET `then`

`defuncts`	$select : wset \times bool \to wset$
	$switch : bool \times bool \to bool$
	$put : entry \times list \times bool \times bool \to list$
	$incr : nat \times bool \times bool \to nat$
`vars`	$x : entry \;\; L : list \;\; n : nat \;\; b, c : bool \;\; W : wset$
`Horn axioms`	$select(W, true) \equiv W$
	$select(W, false) \equiv empty$
	$switch(b, b) \equiv not(b)$
	$switch(b, c) \equiv b \quad \Leftarrow \quad b \not\equiv c$
	$put(x, L, b, b) \equiv L@[x]$
	$put(x, L, b, c) \equiv L \quad \Leftarrow \quad b \not\equiv c$
	$incr(n, b, b) \equiv n + 1$
	$incr(n, b, c) \equiv n \quad \Leftarrow \quad b \not\equiv c$

We claim that ABP satisfies the reachability condition

$$in(s) \equiv [L_0] \wedge \Box AF(p_6(s) \not\equiv empty) \wedge \Box AF(p_9(s) \not\equiv empty) \;\leadsto\; in(s) \equiv empty \wedge out(s) \equiv [L_0].$$

The invariants in the premise ensure that the channels *transmit* and *transAck* are fair and thus all messages and acknowledgements are transferred eventually (cf. Section 4.3).

Part of this example stems from Jensen [51]. However, his net starts out from place 1 in Fig. 10 where the messages are already numbered and transmits them with the number indices and not with Boolean tags. See also [88], Section 27, for nets representing the alternating bit protocol.

By using *select* in the inscriptions of arcs leaving the transitions *transmit* and *transAck*, this function realizes nondeterministic state transitions. The same effect is accomplished by replacing *select* with a **reset arc** pointing to a transition *lose* (first net of Fig. 11) or by introducing a Boolean place (second net of Fig. 11).

In the third net of Fig. 11, the transition transmits W with a probability of 3/5 instead of 1/2 as in the second net. ❏

5.4 Translation of SDL specifications into nets

Systems (on the lowest level called blocks) presented in the 2-dimensional specification language SDL (cf. [16]) consist of channels (also called signal routes) and processes. These can be compiled stepwise into nets by applying the graph grammar rules of Figures 12 and 13.

Rule 1 generates a place for the channel *channel*, which contains an initially empty queue of messages. Rule 2 equips *process* with initially empty input and output places and a place *s_process* for taking up the actual state of *process* (initially $state_0$). Rules 3 and 4 connect *channel* with the input or output place of *process*, respectively. Rule 3 dequeues *channel* if the latest entry is an instance of one of the messages $m_1, \ldots, m_k$. Rule

[8] For expressing this weighted-set comprehension with *filter* (cf. Section 5.1) the predicate *minrev* must be presented as a Boolean function, which, in turn, requires Horn axioms for the complement of *minrev*. The reader is invited to work out the complete specification.

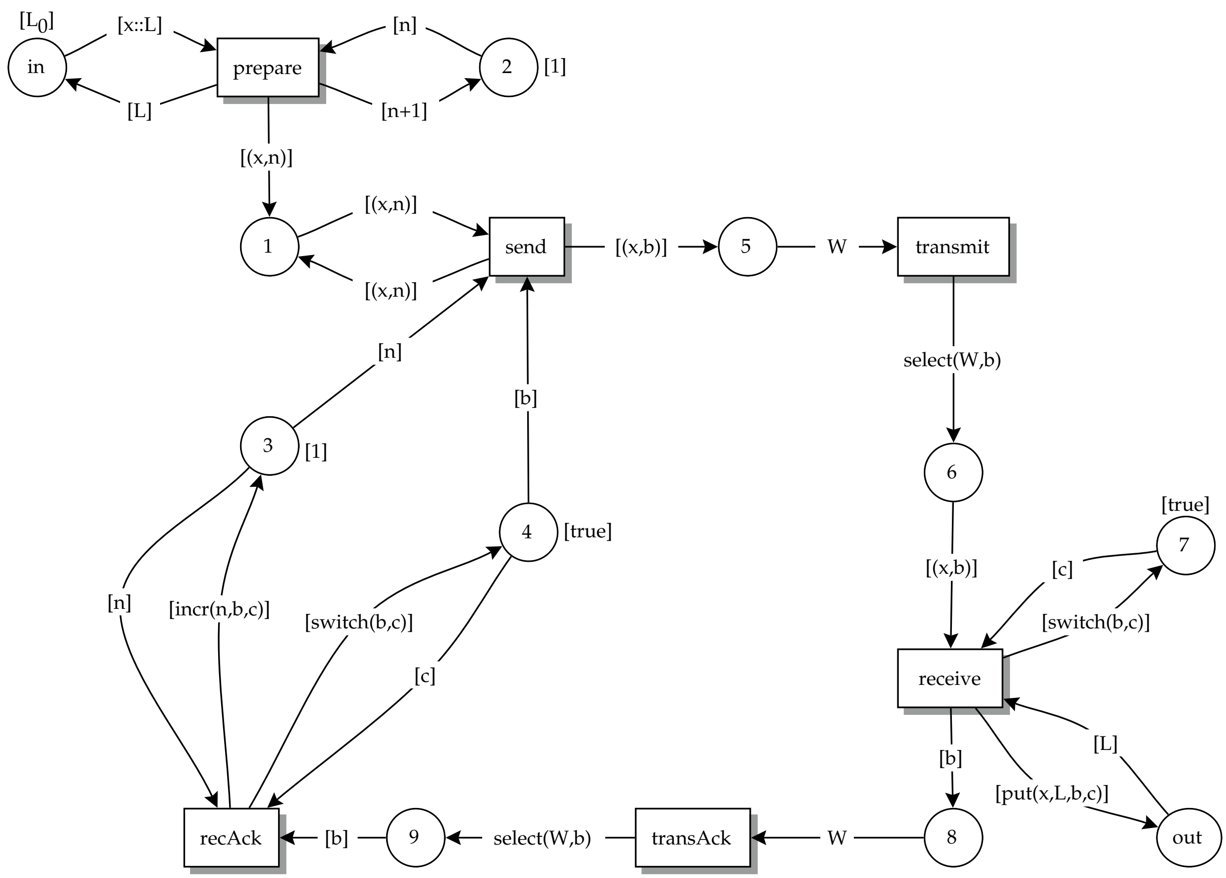

Figure 10. *ABP*

4 enqueues *channel* if the message sent by *process* is an instance of one of the messages $m_1, \ldots, m_k$. Rule 5 creates places for local variables.

Rules 6 through 9 build up the net for the code of *process*. This leads to the generation of transitions for reading from or writing into variable places, dequeueing the input queue and enqueueing the output queue of *process* (which were generated by Rule 2 and 3, respectively). Dequeueing depends on the actual state $state_i$ of *process* (see Rule 6). Big dots denote tasks (compiled by Rule 7), switches (compiled by Rule 8) or message generations (compiled by Rule 9). Dequeueing, task and switch execution are followed by shifting a uniform "control" token *go* to new places produced by further applications of Rules 7, 8 or 9. Rule 9 generates a message, appends it to the output queue, consumes the control token and changes the actual state by putting the new state into the state place of *process* (which was generated and initialized by Rule 4). Dotted arcs denote "gluing points" of a grammar rule.

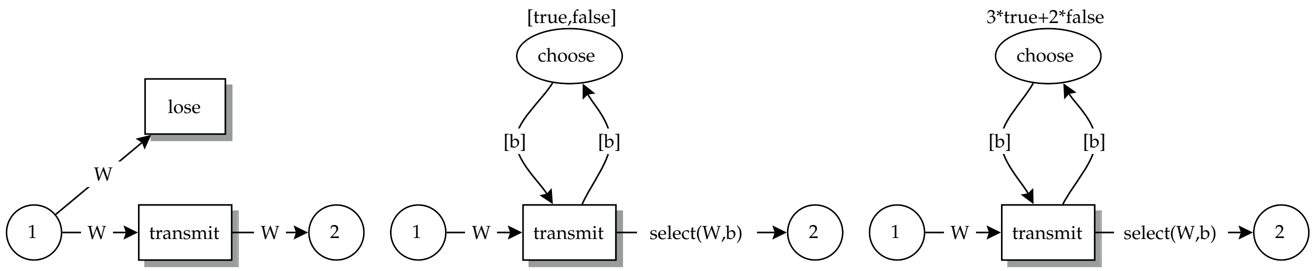

Figure 11. *Net representations of nondeterminism.*

6 Swinging UML

We translate UML class diagrams and state machines [93] into coalgebraic swinging types [79] for making UML models executable and verifiable.

6.1 Class diagrams

A class is denoted by a destructor sort, say cl. An **attribute** $at : s$ of the class takes values in the domain denoted by s and provides a destructor $d : cl \to s$. A **method** (*operation* in UML terminology) $m(x_1 : s_1, \ldots, x_n : s_n)$ of cl is turned into a constructor $c : cl \times s_1 \times \cdots \times s_n \to cl$, while a method $m(x_1 : s_1, \ldots, x_n : s_n) : s$ of cl yields a defined function $f : cl \times s_1 \times \cdots \times s_n \to s$. A **class-scope operation** $m(x_1 : s_1, \ldots, x_n : s_n)$ is translated into a constructor $c : s_1 \times \cdots \times s_n \to cl$. Methods defined in terms of other methods may also be introduced as defined functions.

An n-ary **association** $assoc$ that relates n classes $cl_1, \ldots, cl_n$ is usually regarded as an n-ary relation [19, 62, 90]. Then **rolenames** attached to the ends of $assoc$ correspond to attributes in the sense of relational data models or projections from an algebraic point of view. Since $cl_1, \ldots, cl_n$ are hidden, $assoc$ is also a hidden sort with the membership function $\in: (cl_1 \times \cdots \times cl_n) \times assoc \to bool$ as the destructor. Binary and **anonymous** associations, which provide the pathways for navigating between objects of the associated classes, should be translated differently. For reasoning about navigations the relational view enforces the computation of transitive closures of associations. This has been shown to result in rather tricky and counter-intuitive code [62].

Instead of introducing a relational sort for a binary and anonymous association, a rolename attached to one of its ends becomes destructor of the class cl at the opposite end. If an association end lacks a rolename, we introduce one. The range sort of the destructor, say $d : cl \to s$, depends on the **multiplicity** at the end that holds the rolename corresponding to d. If the multiplicity is 1, then $s = clr$ where clr is the class the rolename is attached to. If the multiplicity is $m..n$, $+$ or $*$, then s is a sum sort: $\coprod_{i=m}^{n} clr^i$, $\coprod_{n>0} clr^n$ or $clr^* = \coprod_{n \in \mathbb{N}} clr^n$, respectively.[9] Hence d assigns a list of clr-objects to each cl-object. The relational view suggests a set rather than a list. However, additional constraints may demand another type of collection like a list or a bag. As long we do not want to prove that cl-objects are behaviorally equivalent, the actual collection type is irrelevant so that we can restrict ourselves to lists as they are given by the above sum sorts. List multiplicities can be turned into set or bag multiplicities by deriving transition predicates from destructors as in the steps from ETREE to EPROCESS and EBAG, respectively (cf. Section 4.6).

In [89], binary associations are also translated into set-valued functions. But the authors do not give a semantics of objects. Hence it is not clear what the elements are the sets consist of. The only adequate interpretation of a class diagram is a behavioral one, such as the final model of a coalgebraic swinging type [79]. In particular, if there are binary associations forming a cycle (like the one in Fig. 14), then the objects of the

[9] $clr^0 =_{def} 1$.

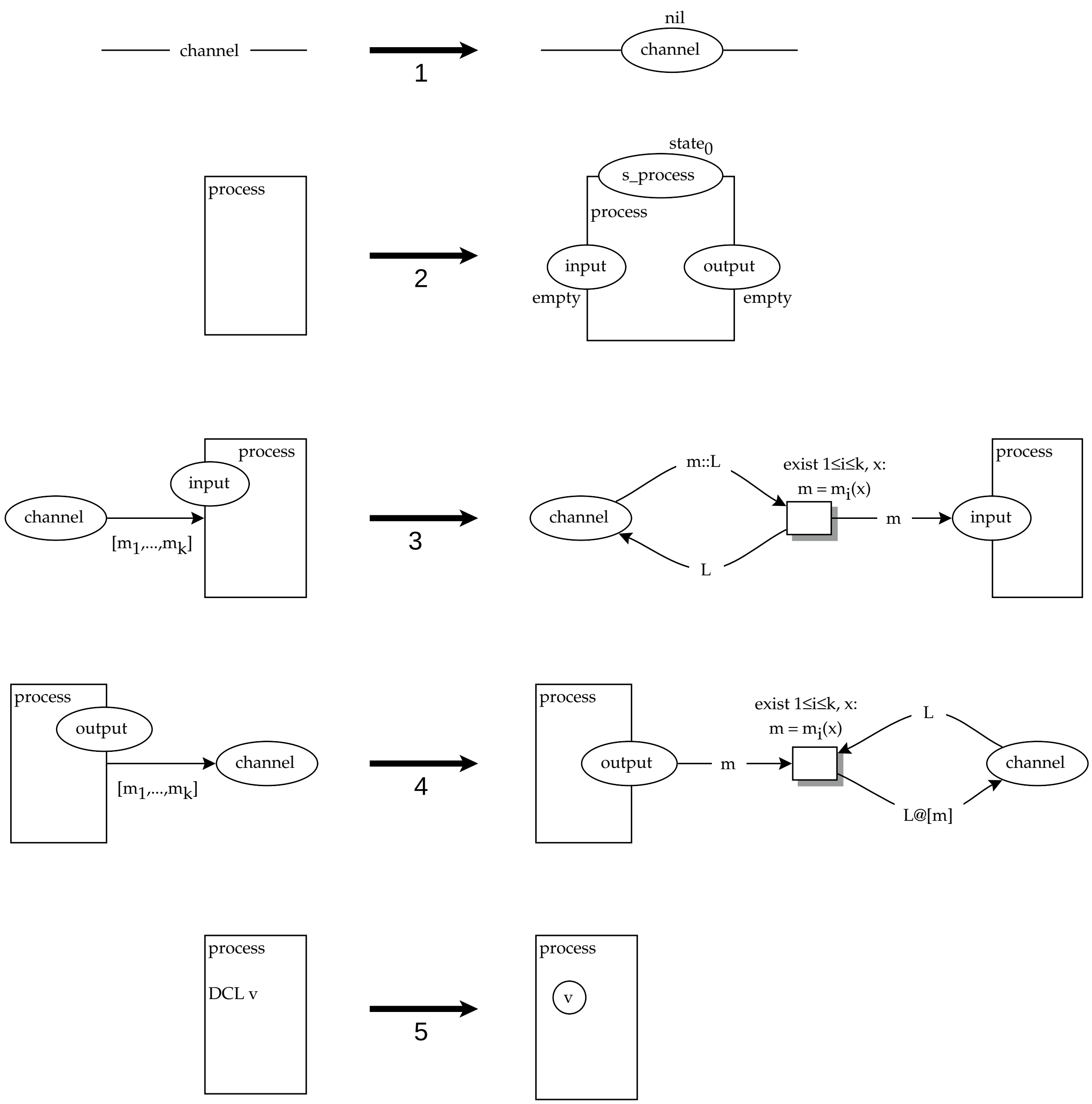

Figure 12. *Translating channels and processes.*

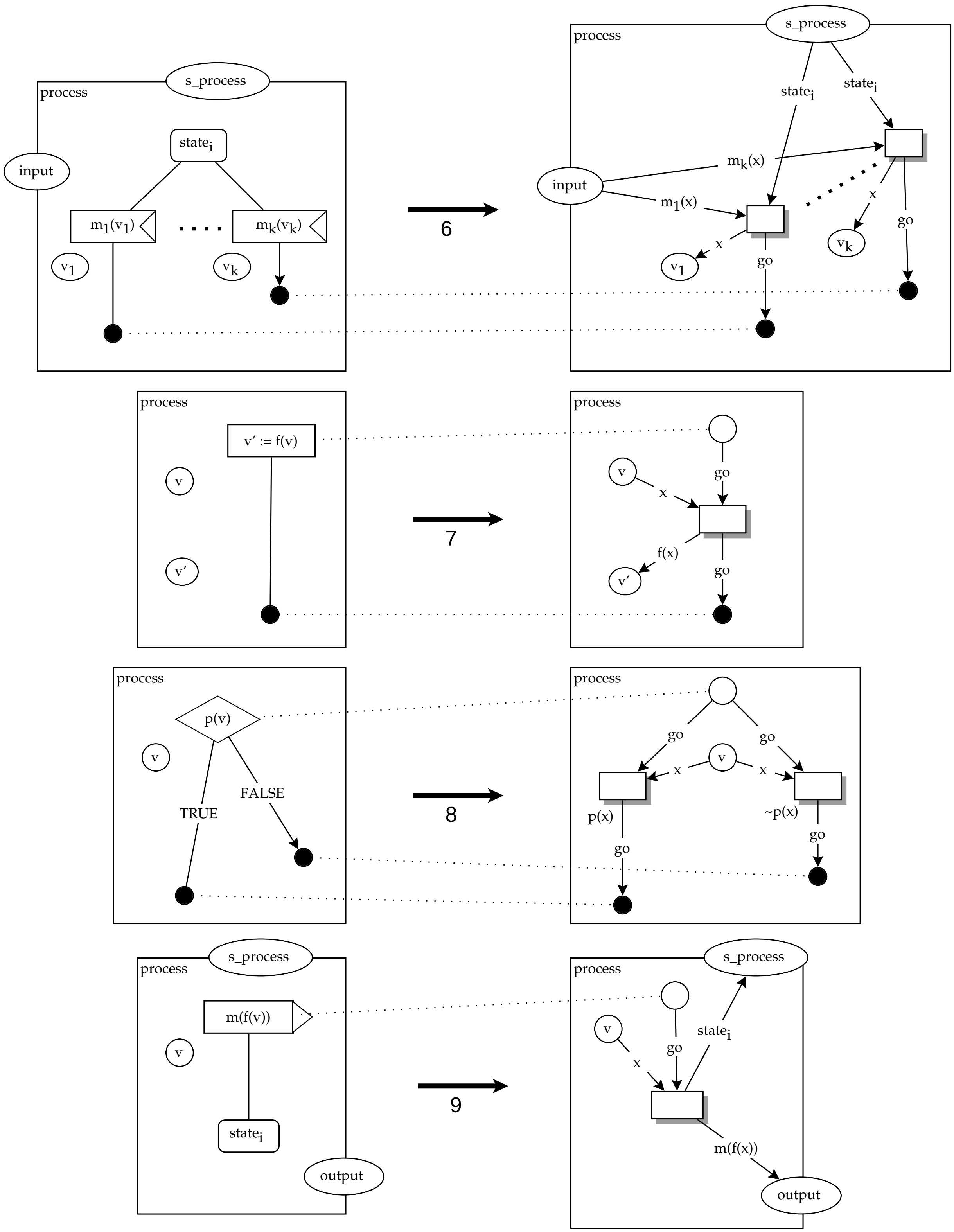

Figure 13. *Translating process code.*

involved classes cannot be built up in a hierarchical way, just by assigning values to attributes and other objects to rolenames. Instead, hidden normal forms in a coalgebraic swinging type denote—often infinite—tuples of functional interpretations of *context* expressions (cf. [79]). Intuitively, such a tuple describes the *behavior* of an object in the state denoted by the normal form. Context expressions are also terms, but they are built up of destructors, here: attributes and rolenames.

The constructors building up hidden normal forms that denote object states may serve different purposes. Either they represent methods or method components and thus reflect the dynamic evolution of objects. Or they reflect the static structure of **composite states**. Or they arise from object **aggregations** or **compositions**. In the last two cases, behavioral equivalence will be inverse compatible with the constructors and thus they can be declared as *object constructors* (cf. Section 1).

Each **generalization** arrow pointing from a subclass cl to a direct superclass cl' of cl yields a constructor $in_{cl}^{cl'} : cl \to cl'$ of the specification $SP(cl)$ of cl. This makes a superclass the sum of itself (if it involves object-creating methods like Java's constructors) and its direct subclasses. Hence generalizations complement **aggregations** that are modelled by products (cf. [23], Section 12.5.2).

Let $<$ be the inheritance relation associated with the class diagram. We assume that the nodes form a finite lattice w.r.t. $<$, i.e., each set of classes of the diagram have a least common superclass. The specification of a class cl in the diagram related by $<$ must be augmented with the following **frame axioms** for defined functions that are passed to cl from sub- or superclasses. Let cl' be a direct subclass of cl.

Given a class cl, two somewhat complementary extensions of $SP(cl)$, the specification of cl, may be necessary. Both extensions are concerned with defined functions $f : cl \times s_1 \times \cdots \times s_n \to s$ that stem from attributes or methods of cl and have already axioms (representing the definition of f within cl) or are at least used in $SP(cl)$. At first, f is equipped with the index cl in order to distinguish f from the synonymous functions in sub- or superclasses of cl. Constructors derived from methods of cl are also indexed with cl.

The first extension reflects the use of cl as a subclass and thus should be applied stepwise, first to the maximal classes w.r.t. $<$ and then downwards from super- to subclasses. The second extension embodies the superclass properties of cl and thus should be applied first to the minimal clases w.r.t. $<$ and then upwards from sub- to superclasses:

- **Adding subclass properties of cl'.** Let $f_{cl} : cl \times s_1 \times \cdots \times s_n \to cl$ and $g_{cl} : cl \times s_1 \times \cdots \times s_n \to s$, $s \neq cl$, be used, but not redefined in $SP(cl')$. Then the following axioms must be added to $SP(cl')$:

$$\begin{aligned} f_{cl'}(x, x_1, \dots, x_n) \equiv y \quad &\Leftarrow \quad f_{cl}(in_{cl'}^{cl}(x), x_1, \dots, x_n) \equiv in_{cl'}^{cl}(y) \\ g_{cl'}(x, x_1, \dots, x_n) \equiv y \quad &\Leftarrow \quad g_{cl}(in_{cl'}^{cl}(x), x_1, \dots, x_n) \equiv y. \end{aligned}$$

- **Adding superclass properties of cl.** Let $f_{cl'} : cl' \times s_1 \times \cdots \times s_n \to cl'$ and $g_{cl'} : cl' \times s_1 \times \cdots \times s_n \to s$, $s \neq cl$, be defined in $SP(cl')$, but not in $SP(cl)$. Then the following axioms must be added to $SP(cl)$:

$$\begin{aligned} f_{cl}(in_{cl'}^{cl}(x), x_1, \dots, x_n) &\equiv in_{cl'}^{cl}(f_{cl'}(x, x_1, \dots, x_n)) \\ g_{cl}(in_{cl'}^{cl}(x), x_1, \dots, x_n) &\equiv g_{cl'}(x, x_1, \dots, x_n). \end{aligned}$$

Multiple inheritance. Let cl be a common superclass of cl_1 and cl_2 and cl_1 and cl_2 be superclasses of cl_3. Then a cl_3-term u may have two normal form representations in cl:

$$t \;=\; in_{cl_1}^{cl}(in_{cl_3}^{cl_1}(u)) \quad \text{and} \quad t' \;=\; in_{cl_2}^{cl}(in_{cl_3}^{cl_2}(u)).$$

For keeping track of the unique origin of cl-terms, $SP(cl)$ is extended by a predicate $\approx_{cl}$: $cl \times cl$ and for each direct subclass cl' of cl, a predicate $\approx_{cl'}^{cl} : cl' \times cl$, specified by the axioms

$$\begin{array}{lcl} x \approx_{cl} in_{cl'}^{cl}(y) & \Leftarrow & x \approx_{cl'}^{cl} y \\ in_{cl'}^{cl}(y) \approx_{cl'}^{cl} z & \Leftarrow & y \approx_{cl'} z \\ x \approx_{cl} x & & \\ x \approx_{cl} y & \Leftarrow & y \approx_{cl} x \\ x \approx_{cl} z & \Leftarrow & x \approx_{cl} y \wedge y \approx_{cl} z. \end{array}$$

Let SP be the union of specifications derived from the class diagram. $\approx_{cl}^{SP} = \approx_{cl}^{Her(SP)}$ is an equivalence relation that identifies all ground cl-terms t, t', which stem from the same term of a subclass of cl, such as u above. Let S be the set of all sorts of SP and CS be the subset of class sorts. If $\approx_{cl}$ shall be used as an equality instead of the structural or behavioral cl-equality, then

$$\approx_{SP} \; = \; \{\approx_{cl}^{SP} \mid cl \in CS\} \cup \{\sim_{s}^{SP} \mid s \in S \setminus CS\}$$

must be made compatible with the functions of SP, i.e., a function $f : cl \times w \rightarrow s$ must be specified in a way such that for all $u \in T_{\Sigma,w}$, $t \approx_{cl}^{SP} t'$ implies $f(t,u) \approx_{SP} f(t',u)$.

Of course, neither the additional axioms for functions nor those for hierarchy-preserving equalities must be added "by hand". They can be constructed automatically and need to be available only when class specifications are actually tested or verified, i.e. SP-formulas are solved or proved.

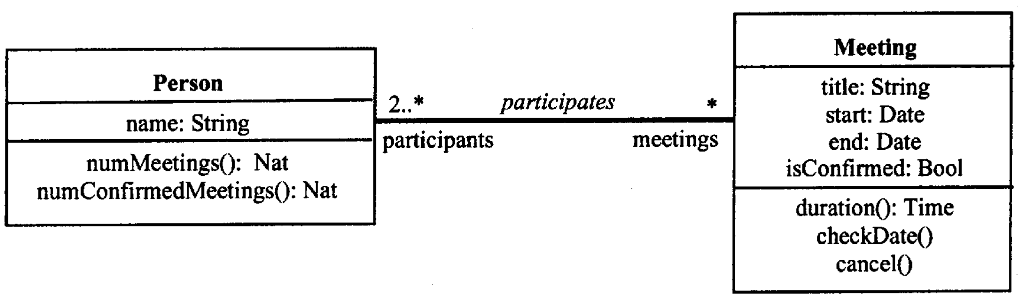

Figure 14. *Two associated classes (Figure 3 of [46]).*

In accordance with [93], classes with **qualifiers** establish product sorts. [29] compiles generalizations and qualifiers into more basic UML concepts. But since the target language is again UML (and OCL), the translation does not take us closer to a verifiable model.

UML classes may be equipped with invariants, associations with multiplicity or Boolean constraints, operations with pre/postconditions. All this is expressed in OCL [102], the "object constraint language" UML is associated with. Multiplicity constraints and invariants restrict the possible behaviors of class instances. Hence they amount to *assertions* of the associated swinging type. Pre/postconditions are translated into Horn axioms for defined functions. Since partially-defined models are too difficult to handle in a formal way, the given pre/postconditions must be "completed" in order to yield total functions in the swinging type's final model. Adequate completions can always be achieved with the help of sum sorts consisting of disjoint "defined" and "undefined" summands. In contrast to other approaches sum sorts minimize the specification overhead that is concerned with partiality. Starting out from the unit sort 1 with its single element (), a detailed exception handling may be postponed to later refinements of the specification. One of the main proof obligations inherent to a constraint-augmented class diagram is to show that the (completed) pre/postconditions respect all class invariants.

A bare class diagram usually contains only a small subset of all desired use relationships between attributes, operations and rolenames. As [46] points out, it is the additional constraints like invariants or pre/postconditions that define "strongly connected" subdiagrams. Each of them covers all attributes and rolenames that occur in some constraint because all of these must be navigated for checking the constraint. For accomplishing an adequate class hierarchy, [46] proposes the refinement of a class diagram in a way that turns the strongly connected subdiagrams into new superclasses (generalizations). This strategy raises the question whether a reasonable grouping of operations into classes can be achieved at all *before* most of the constraints have been fixed. One may plead for hierarchical, parameterized specifications rather than class diagrams in those early design phases where many constraints are not yet known. But this is debatable because class diagrams allow people, as a referee puts it, "to look at parts of the map" without being bothered by a "counter-productive" hierarchy "when the design is still rather vague and prone to changes".

[46] asserts a conceptual difference between the algebraic specification methodology and the object-oriented modelling approach: the former favors if not demands a high degree of data encapsulation and constraint locality, while the latter admits, at least on higher design levels, the "free use of information from almost anywhere in the current system state." On lower levels, the object-oriented approach achieves locality "by enriching the operation lists of the classes and by switching to a message-passing semantics. Sending a message to a locally known object and reading the result may be the equivalent to a complex navigation over the object community—however, the global actions, which are caused by sending a message, are invisible to the invoking object."

It might be a widespread practice in algebraic specification to enforce a high degree of locality and encapsulation, but this is not inherent to the approach. [46] claims a general one-to-one correspondence between a class and a specification unit. However, a simple look at the graph structure of a class diagram reveals that this cannot work as soon as the graph involves cycles such as those created by bidirectional associations (cf. Fig. 14). A class does not correspond to a whole specification, but just to a single *sort*. Due to the "static" semantics, an algebraic specification is structured hierarchically: the use relationships form a collapsed *tree*.

Even the finest specification structure reflecting a class diagram has to encapsulate all data and operations involved in a cycle of associations into a single specification unit. But we need not head for the other extreme—recommended by [46]—and turn the entire class diagram into a single type with a global state sort. This gives up the modularity of object-oriented specifications and thus establishes a semantics far from what the syntax suggests.

Example 6.1.1 The class diagram of Fig. 14 is presented as a coalgebraic swinging type whose axioms cover the multiplicity constraints of Fig. 14 and the following *OCL constraint* [102] taken from [46]:

```
context Meeting :: checkDate()
      post : isConfirmed =
             self.participants ->
             collect(meetings) ->
             forAll(m | m <> self and m.isConfirmed implies
                      (after(self.end,m.start) or (after(m.end,self.start)))
```

(cf. [46], Fig. 4).

PERSON&MEETING = FINSET `and` STRING `and` DATE&TIME `then`

`hidsorts` $Person \;\; Meeting$

`destructs` $name : Person \to String$

$meetings : Person \to Meeting^*$

$title : Meeting \to String$

$participants : Meeting \to Person^*$

$start, end : Meeting \to Date$

	$isConfirmed : Meeting \to bool$
`constructs`	$checkDate : Meeting \to Meeting$
	$cancel : Meeting \to Meeting$
`defuncts`	$Meetings : Person \to set(Meeting)$
	$Participants : Meeting \to set(Person)$
	$numMeetings : Person \to nat$
	$numConfirmedMeetings : Person \to nat$
	$duration : Meeting \to Time$
	$consistent : Meeting \times Meeting \to bool$
`vars`	$p : Person \quad m, m' : Meeting \quad ms : set(Meeting) \quad ps : set(Person)$
`Horn axioms`	$Meetings(p) \equiv mkset(meetings(p))$
	$Participants(m) \equiv mkset(participants(m))$
	$numMeetings(p) \equiv \lvert Meetings(p)\rvert$
	$numConfirmedMeetings(p) \equiv \lvert filter(isConfirmed, Meetings(p))\rvert$
	$duration(m) \equiv end(m) - start(m)$
	$consistent(m, m') \equiv not(isConfirmed(m'))$
	$\quad or\ end(m) < start(m')\ or\ end(m') < start(m)$
	$isConfirmed(checkDate(m)) \equiv forall(\lambda m'.consistent(m, m'), remove(m, ms))$
	$\Leftarrow Participants(m) \equiv ps \wedge flatten(map(Meetings, ps)) \equiv ms$
	$isConfirmed(cancel(m)) \equiv false$
`assertions`	$\lvert Participants(m)\rvert \geq 2$

Classes come as hidden sorts, attributes and roles as destructors, roles usually as non-linear ones. Basic methods are declared as constructors, derived ones as defined functions. Let CSP be the cospecification of PERSON&MEETING. The elements of $Fin(CSP)_{Person}$ and $Fin(CSP)_{Meeting}$ may be visualized as infinite trees whose edges represent the relation between object states that is induced by the *participates* association of Fig. 14. The UML semantics of Fig. 14 requires sets rather than sequences as values of *meetings* and *participants*. This is reflected by the fact that all axioms of PERSON&MEETING do not use these destructors directly, but only their set versions *Meetings* and *Participants*. ❏

Example 6.1.2 The class diagram of Fig. 15 leads to the following (incomplete) swinging type. The $*$-multiplicity at the bottom of the diagram is turned into a more reasonable 1-multiplicity. The generalization induces additional constructors $in^{Res}_{SS} : SS \to Reservation$ and $in^{Res}_{IR} : IR \to Reservation$ (see above). $SP[s_1, \ldots, s_n]$ indicates that SP is a parameter specification that provides sorts $s_1, \ldots, s_n$.

RESERVATION = FINSET `and` STRING `and` TIME `and` INT `then`

`hidsorts`	$Customer \quad Reservation \quad SS \quad IR \quad Ticket \quad Show \quad Performance$	
	$CustomId = String \quad TicketId = Date \times (int + 1) \times String \times TimeOfDay$	
`constructs`	$new : CustomId \times String \to Customer$	corresponds to add(name,phone) in Fig. 15
	$new : TicketId \to Ticket$	
	$new : Date \times TimeOfDay \to Performance$	
	$new : int \to SS$	
	$new : \to IR$	
	$buy : Customer \times Ticket \to Customer$	
	$sell : Ticket \times Customer \to Ticket$	
	$exchange : Ticket \to Ticket$	left unspecified
	$in^{Res}_{SS} : SS \to Reservation$	
	$in^{Res}_{IR} : IR \to Reservation$	
`destructs`	$name : Customer \to String$	

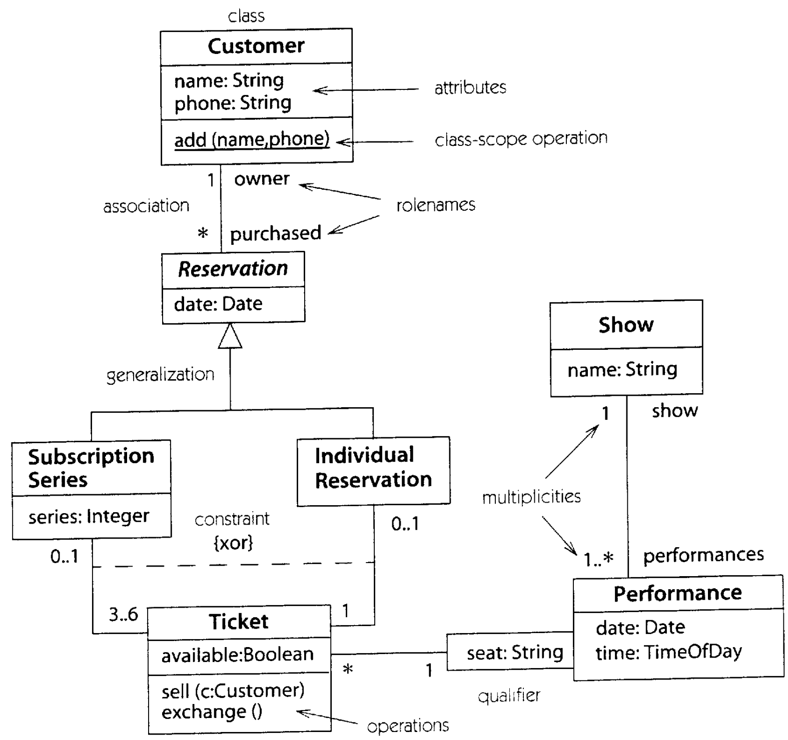

Figure 15. *A "big" class diagram (Figure 3-1 of [93]).*

	$phone : Customer \to String$	
	$purchased : Customer \to Reservation^*$	
	$date : Reservation \to Date$	
	$owner : Reservation \to Customer$	
	$series : SS \to int$	
(1)	$tickets : SS \to Ticket^*$	
(2)	$ticket : IR \to Ticket$	
	$available : Ticket \to bool$	
	$ID : Ticket \to Date \times (int + 1) \times String \times TimeOfDay$	
(3)	$subscriptionSeries : Ticket \to 1 + SS$	
(4)	$individualReservation : Ticket \to 1 + IR$	
(5)	$qperformance : Ticket \to String \times Performance$	qualified performance
	$name : Show \to String$	
	$performances : Show \to Performance^*$	
	$date : Performance \to Date$	
	$time : Performance \to TimeOfDay$	
	$show : Performance \to Show$	
	$seat : String \times Performance \to String$	
(6)	$ticket : String \times Performance \to Ticket$	
`vars`	$c : Customer \quad cid : CustomId \quad t : Ticket \quad tid : TicketId \quad pho, sea : String \quad sh : Show$	
	$ss : SS \quad ir : IR \quad b : bool \quad i : int \quad dat : Date \quad tim : TimeOfDay \quad p : Performance$	
	$r_1, \dots, r_n : Reservation$	
`Horn axioms`	$name(new(cid, pho)) \equiv cid$	
	$phone(new(cid, pho)) \equiv pho$	
	$purchased(new(cid, pho)) \equiv ()$	

(A)	$name(buy(c,t)) \equiv name(c)$
(B)	$phone(buy(c,t)) \equiv phone(c)$
	$purchased(buy(c,t)) \equiv (in_{SS}^{Res}(ss), r_1, \dots, r_n)$
	$\Leftarrow\ subscriptionSeries(t) \equiv (ss) \wedge purchased(c) \equiv (r_1, \dots, r_n)$
	$purchased(buy(c,t)) \equiv (in_{IR}^{Res}(ir), r_1, \dots, r_n)$
	$\Leftarrow\ individualReservation(t) \equiv (ir) \wedge purchased(c) \equiv (r_1, \dots, r_n)$
	$available(new(tid)) \equiv true$
	$ID(new(tid)) \equiv tid$
	$subscriptionSeries(new(dat,(i),sea,tim)) \equiv (new(i))$
	$subscriptionSeries(new(dat,(),sea,tim)) \equiv ()$
	$individualReservation(new(dat,(i),sea,tim)) \equiv ()$
	$individualReservation(new(dat,(),sea,tim)) \equiv (new)$
	$qperformance(new(dat,(i),sea,tim)) \equiv (sea, new(date,time))$
	$available(sell(t,c)) \equiv false$
(C)	$ID(sell(t,c)) \equiv ID(t)$
(D)	$subscriptionSeries(sell(t,c)) \equiv subscriptionSeries(t)$
(E)	$individualReservation(sell(t,c)) \equiv individualReservation(t)$
(F)	$qperformance(sell(t,c)) \equiv qperformance(t)$
	$seat(sea,p) \equiv sea$
`assertions`	$3 \leq \lvert mkset(tickets(ss)) \rvert \leq 6$
	$subscriptionSeries(t) \equiv () \ \dot{\vee}\ individualReservation(t) \equiv ()$
	$\lvert mkset(performances(sh)) \rvert > 0$

All operations of Fig. 15 are declared as constructors. The methods *new* and *buy* were added for making the example a little more complete. The Horn axioms describe the operations' pre/postconditions. Further preconditions are part of the state machine that may be associated with a class diagram (cf. Ex. 6.2.5). The destructors (1)-(6) represent the anonymous association ends in Fig. 15. (A)-(F) are **frame axioms** expressing that certain operations do not affect the values of certain attribute or rolenames. After the actual effects of all operations have been specified, frame axioms can be added automatically. ❏

For referring to individual objects **object identifiers** (address, name, number, etc.) or **key attributes** in the sense of relational data bases must be distinguished among the attributes of a class (cf., e.g., [19]). Hence object identifiers are (tuples of) particular attributes. In Ex. 6.1.1, the destructors $name : Person \to String$ and $title : Meeting \to String$ are the object identifiers of $Person$ resp. $Meeting$. In Ex. 6.1.2, the destructors $name : Customer \to String$ and $ID : Ticket \to Date \times (int+1) \times String \times TimeOfDay$ are the object identifiers of $Customer$ resp. $Ticket$.

6.2 State machines

UML uses **state machines** for specifying operations like *sell* and *buy* (cf. Ex. 6.1.2). State machines are labelled transition systems that adopt (part of) the *statechart* approach [41].[10] A transition from state st_1 to state st_2 may have many components:

$$st_1 \xrightarrow{e(x)[g(x)]/act(x)} st_2.$$

The transition is caused by a parameterized **event** $e(t)$ if the **guard** (= Boolean expression) $g(x)$ applied to t evaluates to true. During the state transition, the **action** $act(t)$ is executed. UML distinguishes between several kinds of events and actions.

[10] For differences between statecharts and state machines concerning the semantics of synchronization, see [58], Section 2.4.

More precisely, the event $e(t)$ denotes a message that is received by an object in state st_1, while the action $act(t)$ denotes a message that is sent by an object in state st_1. The same transition label *lab* may denote an event at some time and an action at another time. The actual rôle of *lab* depends on the sort of st_1, i.e. on the class whose objects may send or receive *lab*. Objects of the same class can treat *lab* only *either* as an action *or* as an event.

The crucial point in a formal semantics of state machines is the notion of a *state* and what it is to represent. [102] regards states as values of a particular attribute. What distinguishes state attributes from other attributes? That they can take only finitely many values so that state machines are representable as graphs? Are all state attributes determined by class attributes? If so, it would be reasonable to define states as tuples of values over all attributes of a class. Consequently, state sets will usually be infinite.

[37] associates states with predicates whose validity may change when transitions take place. Then there should be some guidelines telling us which predicates form states and which ones form guards. At some stage of a refinement of the state model, the predicate denoting a state s should imply the guards that label transitions starting out from s. States-as-predicates realize the *two-tiered view* of modal logic and Kripke models: a state is a world, state transitions change worlds, the structure of states and the structure of transition systems are expressed on different levels in different languages. Alternatively, process algebra [7], dynamic data types [6], hidden algebra [30, 32] and swinging types proclaim the *one-tiered view* where states and state transitions pertain to the same world.

UML uses state machines only as a *description* tool. If they shall provide the models against which system properties are to be *verified*, the choice between the one- and the two-tiered view becomes crucial. We consider the former to be more adequate especially when—as in UML state machines—events and actions are calls of functions that belong to the static part of a system.

STs represent states as hidden-sorted *normal forms*, i.e., terms built up of constructors. In the beginning of a system development, a normal form may represent a state uniquely (cf., e.g., Exs. 6.2.2 and 6.2.3 and the *resource*-constructors in Ex. 6.2.4). In later stages, a state may obtain many, though *behaviorally equivalent*, normal form representations. Normal forms consisting of *object* constructors (cf. Sect. 2) are behaviorally equivalent *only if* they are equal. However, *objects* cannot be identified by the normal forms representing their possible states. Their identity is determined by object identifiers (see above).

State transitions should preserve behavioral equivalence, which makes this relation into a *bisimulation*: it is not fully, but only *zigzag* compatible with transitions. This means that the ability of a transition to be executed and the result of the execution do not depend on the source state's term representation. Full versus zigzag compatibility motivates the separation of static predicates from dynamic ones.

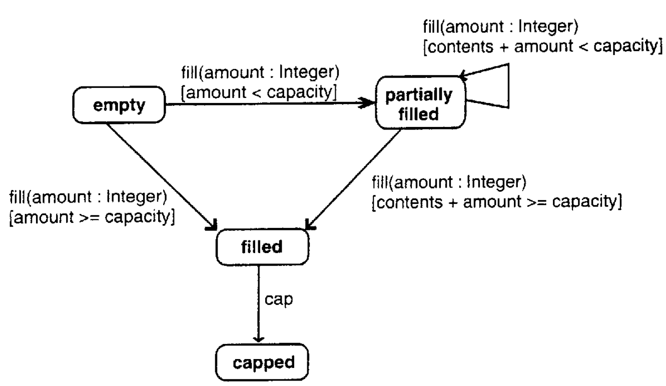

Figure 16. *A state machine with events and guards, but no actions (Figure 4-4 of [102]).*

Example 6.2.1 In Section 3.1, we have presented several variants of a 3-level bank trade specification, which employed the states-as-objects approach. If we apply the schema underlying the second variant ACCOUNTS2 to the state machine of Fig. 16, we come up with a corresponding 3-level specification. The sort *com* is replaced by the sort *event*, and *sets* of accounts are replaced by *bags* of bottles because here we do not need unique object identifications.

BOTTLESTATE = NAT **then**

`hidsorts`	$Bottle$
`constructs`	$new : nat \to Bottle$
	$fill : Bottle \times nat \to Bottle$
	$cap : Bottle \to Bottle$
`destructs`	$capacity, contents : Bottle \to nat$
	$capped : Bottle$
`static preds`	$empty, filled, partially\_filled : Bottle$
`vars`	$n : nat \quad b : Bottle$
`Horn axioms`	$capacity(new(n)) \equiv n$
	$contents(new(n)) \equiv 0$
	$uncapped(new(n))$
	$capacity(fill(b,n)) \equiv capacity(b)$
	$contents(fill(b,n)) \equiv min(contents(b)+n, capacity(b))$
	$capped(cap(b))$
	$capacity(cap(b)) \equiv capacity(b)$
	$contents(cap(b)) \equiv contents(b)$
	$empty(b) \quad \Leftarrow \quad contents(b) \equiv 0$
	$filled(b) \quad \Leftarrow \quad contents(b) \equiv capacity(b)$
	$partially\_filled(b) \quad \Leftarrow \quad 0 < contents(b) < capacity(b)$

The specification reveals that each state in Fig. 16 represents a set of attribute (destructor) values. Hence there is no need for an additional state attribute (cf. [102], Section 4.1.5). The example also shows that state predicates and transition guards encompass similar semantical information. Both provide preconditions for executing a transition.

BOTTLETRANS = BOTTLESTATE **then**

`sorts`	$event$
`constructs`	$Fill : nat \to event$
	$Cap : \to event$
`dynamic preds`	$\_ \stackrel{-}{\longrightarrow} \_ : Bottle \times event \times Bottle$
`Horn axioms`	$b \stackrel{Fill(n)}{\longrightarrow} fill(b,n) \quad \Leftarrow \quad empty(b)$
	$b \stackrel{Fill(n)}{\longrightarrow} fill(b,n) \quad \Leftarrow \quad partially\_filled(b)$
	$b \stackrel{Cap}{\longrightarrow} cap(b) \quad \Leftarrow \quad filled(b)$

$Fill(n)$ and Cap are the events that cause transitions from a bottle state b to the state $fill(b,n)$ resp. $cap(b)$.

BOTTLES = BOTTLETRANS **and** FINSET **then**

`constructs`	$New : nat \to event$
`dynamic preds`	$\_ \stackrel{-}{\Longrightarrow} \_ : set(Bottle) \times event \times set(Bottle)$
`vars`	$b, b' : Bottle \quad bs : set(Bottle) \quad e : event$
`Horn axioms`	$bs \stackrel{New(n)}{\Longrightarrow} insert(new(n), bs)$
	$bs \stackrel{e}{\Longrightarrow} insert(b', remove(b, bs)) \quad \Leftarrow \quad b \in bs \wedge b \stackrel{e}{\longrightarrow} b'$

In terms of UML, events like $New(n)$, which trigger concurrent transitions between distributed states of several objects, execute **class-scope operations**. ❏

The term representation of states allows us to identify a state with an **entry action** (UML notion) to be executed when the state is entered.

There are two kinds of state constructors: object generators like *new* and object modifiers like *fill* and *cap*. Object generators have a hidden-sorted range, while object modifiers have a distinguished hidden-sorted argument and a range of the same sort. **Composite states** are built up of *non-recursive* object generators, i.e. the sort of a composite state differs from the sorts of its substates.

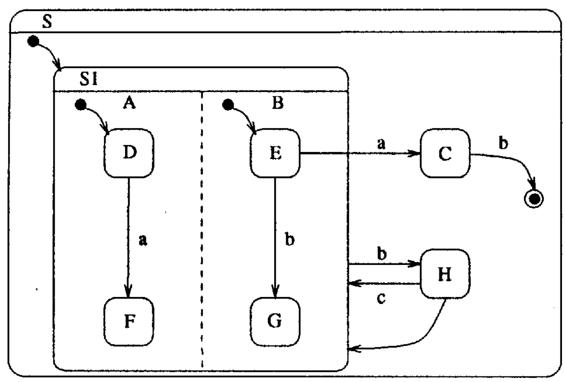

Figure 17. *A state machine with composite states, normal and abnormal exits (Figure 2 of [58]).*

Example 6.2.2 The state machine of Fig. 17 implicitly involves four hidden sorts *outer*, *middle*, A and B and may be translated into a swinging type that follows the one given in [58], Fig. 3:[11]

ALPHABET

`sorts`	$event$	
`hidsorts`	$outer \quad middle \quad A \quad B$	
`constructs`	$a, b, c :\rightarrow event$	
	$:\rightarrow event$	This event triggers *normal completion transitions*.
`objconstructs`	$S : middle \rightarrow outer$	
	$S1 : A \times B \rightarrow middle$	
	$C, H, final :\rightarrow middle$	
	$D, F :\rightarrow A$	
	$E, G :\rightarrow B$	
`dynamic preds`	$\_ \stackrel{\_}{\longrightarrow} \_ : outer \times event \times outer$	
`vars`	$x : A \quad y : B \quad z : middle$	
`Horn axioms`	$S(S1(D,y)) \stackrel{a}{\longrightarrow} S(S1(F,y))$	
	$S(S1(x,E)) \stackrel{b}{\longrightarrow} S(S1(x,G))$	
	$S(S1(x,E)) \stackrel{a}{\longrightarrow} S(C)$	
	$S(C) \stackrel{b}{\longrightarrow} S(final)$	
	$S(S1(x,y)) \stackrel{b}{\longrightarrow} S(H)$	
	$S(H) \stackrel{c}{\longrightarrow} S(S1(D,E))$	
	$S(H) \longrightarrow S(S1(D,E))$	

Note that all axioms are coinductive. ❏

Example 6.2.3 The state machine of Fig. 18 is translated analogously:

[11] The thread denotations A and B become sorts and not constructors.

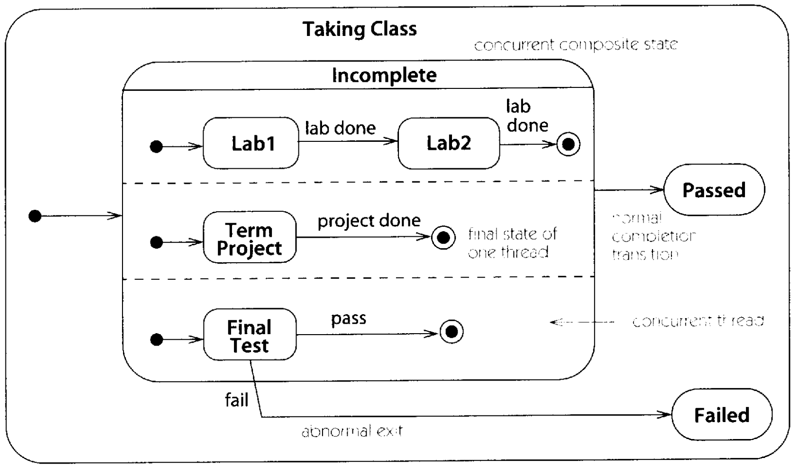

Figure 18. *A state machine with composite states (Figure 6-6 of [93]).*

TAKING_CLASS

`sorts`	$event$
`hidsorts`	$outer \quad A \quad B \quad C$
`constructs`	$lab\_done, project\_done, pass, fail :\rightarrow event$
	$:\rightarrow event$
`objconstructs`	$Incomplete : A \times B \times C \rightarrow outer$
	$Passed, Failed :\rightarrow outer$
	$Lab1, Lab2, finalA :\rightarrow A$
	$Term\_Project, finalB :\rightarrow B$
	$Final\_Test, finalC :\rightarrow C$
`dynamic preds`	$\_ \stackrel{-}{\longrightarrow} \_ : outer \times event \times outer$
`vars`	$x : A \quad y : B \quad z : C$
`Horn axioms`	$Incomplete(Lab1, y, z) \stackrel{lab\_done}{\longrightarrow} Incomplete(Lab2, y, z)$
	$Incomplete(Lab2, y, z) \stackrel{lab\_done}{\longrightarrow} Incomplete(finalA, y, z)$
	$Incomplete(x, Term\_Project, z) \stackrel{project\_done}{\longrightarrow} Incomplete(x, finalB, z)$
	$Incomplete(x, y, Final\_Test) \stackrel{pass}{\longrightarrow} Incomplete(x, y, finalC)$
	$Incomplete(x, y, Final\_Test) \stackrel{fail}{\longrightarrow} Failed$
	$Incomplete(finalA, finalB, finalC) \longrightarrow Passed$

Note that all axioms are coinductive. ❏

Example 6.2.4 (mutual exclusion) The following ST specifies the mutually-exclusive access of several agents to a single resource. As in Example 6.2.1, we begin with "static specifications" of hidden sorts and proceeds with single-object transitions and, finally, multiple-object transitions.

MUTEX = NAT `and` BOOL `and` FINSET `then`

`sorts`	$event \quad label$
`hidsorts`	$agent \quad resource \quad environment = set(agent) \times resource$
`constructs`	$free, used :\rightarrow resource$
	$new : int \rightarrow agent$
	$access, release : agent \rightarrow agent$
	$New : nat \rightarrow event$
	$Request, Access, Release :\rightarrow event$

	$\_.\_ : int \times event \to event$
	$\_ : event \to label$
	$\_/\_ : event \times event \to label$
destructs	$id : agent \to nat$
	$uses : agent \to bool$
static preds	$mutex : environment$
dynamic preds	$\_ \stackrel{\_}{\longrightarrow} \_ : agent \times label \times agent$
	$\_ \stackrel{\_}{\longrightarrow} \_ : resource \times label \times resource$
	$\_ \stackrel{\_}{\Longrightarrow} \_ : set(agent) \times label \times set(agent)$
	$\_ \Longrightarrow \_ : environment \times environment$
ν-preds	$\Box mutex : environment$
vars	$e, act : event \quad i : nat \quad a, a' : agent \quad r, r' : resource \quad s, s' : set(agent)$
	$env, env' : environment$
Horn axioms	$id(new(i)) \equiv i$
	$id(access(a)) \equiv id(a)$
	$id(release(a)) \equiv id(a)$
	$uses(new(i)) \equiv false$
	$uses(access(a)) \equiv true$
	$uses(release(a)) \equiv false$
(A)	$a \stackrel{Request}{\longrightarrow} a$
(B)	$a \stackrel{Access}{\longrightarrow} access(a)$
(C)	$a \stackrel{Release}{\longrightarrow} release(a) \quad \Leftarrow \quad uses(a) \equiv true$
(D)	$free \stackrel{i.Request/i.Access}{\longrightarrow} used$
(E)	$used \stackrel{Release}{\longrightarrow} free$
(F)	$s \stackrel{New(i)}{\Longrightarrow} insert(new(i), s)$
(G)	$s \stackrel{i.e}{\Longrightarrow} insert(a', remove(a, s)) \quad \Leftarrow \quad a \in s \ \wedge \ a \stackrel{e}{\longrightarrow} a' \ \wedge \ id(a) \equiv i$
(H)	$(s, r) \Longrightarrow (s', r') \quad \Leftarrow \quad s \stackrel{e}{\Longrightarrow} s' \ \wedge \ r \stackrel{e/act}{\longrightarrow} r'$
(I)	$(s, r) \Longrightarrow (s', r') \quad \Leftarrow \quad r \stackrel{e/act}{\longrightarrow} r' \ \wedge \ s \stackrel{act}{\Longrightarrow} s'$
(J)	$(s, r) \Longrightarrow (s', r') \quad \Leftarrow \quad s \stackrel{i.e}{\Longrightarrow} s' \ \wedge \ r \stackrel{e}{\longrightarrow} r'$
(K)	$(s, r) \Longrightarrow (s', r) \quad \Leftarrow \quad s \stackrel{New(i)}{\Longrightarrow} s'$
	$mutex(s, r) \quad \Leftarrow \quad \lvert filter(uses, s) \rvert \leq 1$
co-Horn axioms	$\Box mutex(env) \quad \Rightarrow \quad mutex(env)$
	$\Box mutex(env) \quad \Rightarrow \quad (env \Longrightarrow env' \quad \Rightarrow \quad \Box mutex(env'))$

Axioms H,I,J,K establish communications between an agent and the resource.

The requirement to MUTEX is an invariant: the resource is never accessed by two agents simultaneously, formally: $\Box mutex(\emptyset, free)$. We show that this formula is an inductive theorem of MUTEX. For the rules applied here, see [76, 78]. Kommas separate the factors of a conjunction from each other. The factors of some conjunctions are numbered. At the point where a numbered factor is going to be expanded the number is typed in boldface.

$\Box mutex(\emptyset, free)$

$\Box$-coinduction (cf. Section 5.3)

$\vdash \ \exists q : q(\emptyset, free) \ \wedge \ (q(env) \Rightarrow mutex(env)) \ \wedge \ ((env \Longrightarrow env' \wedge q(env)) \Rightarrow q(env'))$

define q by the axioms $q(s, free) \Leftarrow \lvert filter(uses, s) \rvert \equiv 0$ and $q(s, used) \Leftarrow \lvert filter(uses, s) \rvert \leq 1$,

unfold q and $mutex$

$\vdash \ \lvert filter(uses, \emptyset) \rvert \equiv 0,$

$$((env \equiv (s, free) \wedge |filter(uses, s)| \equiv 0) \;\vee\; (env \equiv (s, used) \wedge |filter(uses, s)| \leq 1))$$
$$\Rightarrow \; \exists\, s, r : (env \equiv (s, r) \wedge |filter(uses, s)| \leq 1),$$
$$(env \Longrightarrow env' \wedge q(env)) \Rightarrow q(env')$$

expansion with $|filter(uses, \emptyset)| \equiv 0$, Boolean rules and variable elimination

$$\vdash \; (env \equiv (s, free) \wedge |filter(uses, s)| \equiv 0) \;\Rightarrow\; \exists\, s, r : (env \equiv (s, r) \wedge |filter(uses, s)| \leq 1),$$
$$(env \equiv (s, used) \wedge |filter(uses, s)| \leq 1) \;\Rightarrow\; \exists\, s, r : (env \equiv (s, r) \wedge |filter(uses, s)| \leq 1),$$
$$(env \Longrightarrow env' \wedge q(env)) \Rightarrow q(env')$$

quantor elimination

$$\vdash \; (env \equiv (s, free) \wedge |filter(uses, s)| \equiv 0) \;\Rightarrow\; (env \equiv (s, free) \wedge |filter(uses, s)| \leq 1),$$
$$(env \equiv (s, used) \wedge |filter(uses, s)| \leq 1) \;\Rightarrow\; (env \equiv (s, used) \wedge |filter(uses, s)| \leq 1),$$
$$(env \Longrightarrow env' \wedge q(env)) \Rightarrow q(env')$$

Boolean rules

$$\vdash \; (env \Longrightarrow env' \wedge q(env)) \Rightarrow q(env')$$

unfold $\Longrightarrow$ (with axioms H,I,J,K)

$$\vdash \; (((env \equiv (s, r) \wedge env' \equiv (s', r') \wedge s \overset{e}{\Longrightarrow} s' \wedge r \xrightarrow{e/act} r') \vee$$
$$(env \equiv (s, r) \wedge env' \equiv (s', r') \wedge r \xrightarrow{e/act} r' \wedge s \overset{act}{\Longrightarrow} s') \vee$$
$$(env \equiv (s, r) \wedge env' \equiv (s', r') \wedge s \overset{i.e}{\Longrightarrow} s' \wedge r \xrightarrow{e} r') \vee$$
$$(env \equiv (s, r) \wedge env' \equiv (s', r) \wedge s \overset{New(i)}{\Longrightarrow} s')) \;\wedge\; q(env))$$
$$\Rightarrow \; q(env')$$

Boolean rules and variable elimination

$$\vdash \; (s \overset{e}{\Longrightarrow} s' \wedge r \xrightarrow{e/act} r' \wedge q(s, r)) \Rightarrow q(s', r'), \tag{\textbf{1}}$$
$$(r \xrightarrow{e/act} r' \wedge s \overset{act}{\Longrightarrow} s' \wedge q(s, r)) \Rightarrow q(s', r'), \tag{2}$$
$$(s \overset{i.e}{\Longrightarrow} s' \wedge r \xrightarrow{e} r' \wedge q(s, r)) \Rightarrow q(s', r'), \tag{3}$$
$$(s \overset{New(i)}{\Longrightarrow} s' \wedge q(s, r)) \Rightarrow q(s', r) \tag{4}$$

unfold $\Longrightarrow$ (with axioms F,G)

$$\vdash \; (((e \equiv New(i) \wedge s' \equiv insert(new(i), s)) \vee$$
$$(e \equiv i.f \wedge s' \equiv insert(a', remove(a, s)) \wedge a \in s \wedge a \xrightarrow{f} a' \wedge id(a) \equiv i))$$
$$\wedge\; r \xrightarrow{e/act} r' \wedge q(s, r))$$
$$\Rightarrow \; q(s', r'),$$
$$\ldots$$

Boolean rules and variable elimination

$$\vdash \; (r \xrightarrow{New(i)/act} r' \wedge q(s, r)) \;\Rightarrow q(insert(new(i), s), r'), \tag{\textbf{5}}$$
$$(a \in s \wedge a \xrightarrow{f} a' \wedge id(a) \equiv i \wedge r \xrightarrow{i.f/act} r' \wedge q(s, r)) \Rightarrow q(insert(a', remove(a, s)), r'), \tag{6}$$
$$\ldots$$

unfold $\longrightarrow$ (with axioms D,E)

$$\vdash \; (False \wedge q(s, r)) \Rightarrow q(insert(new(i), s), r'),$$
$$\ldots$$

Boolean rules

$$\vdash \; (a \in s \wedge a \xrightarrow{f} a' \wedge id(a) \equiv i \wedge r \xrightarrow{i.f/act} r' \wedge q(s, r)) \Rightarrow q(insert(a', remove(a, s)), r'), \tag{\textbf{6}}$$
$$\ldots$$

variable elimination

$$\vdash \; (a \in s \wedge a \xrightarrow{f} a' \wedge r \xrightarrow{id(a).f/act} r' \wedge q(s, r)) \Rightarrow q(insert(a', remove(a, s)), r'),$$
$$\ldots$$

unfold $\longrightarrow$ (with axioms A,B,C)

$$\vdash \; (a \in s \wedge$$
$$((f \equiv Request \wedge a' \equiv a) \;\vee\; (f \equiv Access \wedge a' \equiv access(a)) \vee$$

$$(f \equiv \mathit{Release} \wedge a' \equiv \mathit{release}(a) \wedge \mathit{uses}(a) \equiv \mathit{true}))$$
$$\wedge\ r \stackrel{id(a).f/act}{\longrightarrow} r' \wedge q(s,r))$$
$$\Rightarrow\ q(\mathit{insert}(a', \mathit{remove}(a,s)), r'),$$
...

Boolean rules and variable elimination

$$\vdash\ (a \in s \wedge r \stackrel{id(a).Request/act}{\longrightarrow} r' \wedge q(s,r)) \Rightarrow q(\mathit{insert}(a, \mathit{remove}(a,s)), r'), \quad \textbf{(7)}$$
$$(a \in s \wedge r \stackrel{id(a).Access/act}{\longrightarrow} r' \wedge q(s,r)) \Rightarrow q(\mathit{insert}(\mathit{access}(a), \mathit{remove}(a,s)), r'), \quad \textbf{(8)}$$
$$(a \in s \wedge \mathit{uses}(a) \equiv \mathit{true} \wedge r \stackrel{id(a).Release/act}{\longrightarrow} r' \wedge q(s,r)) \Rightarrow q(\mathit{insert}(\mathit{release}(a), \mathit{remove}(a,s)), r'), \quad \textbf{(9)}$$
...

unfold $\longrightarrow$ (with axioms D,E)

$$\vdash\ (a \in s \wedge r \equiv \mathit{free} \wedge \mathit{act} \equiv id(a).\mathit{Access} \wedge r' \equiv \mathit{used} \wedge q(s,r)) \Rightarrow q(\mathit{insert}(a, \mathit{remove}(a,s)), r'),$$
$$(a \in s \wedge \mathit{False} \wedge q(s,r)) \Rightarrow q(\mathit{insert}(\mathit{access}(a), \mathit{remove}(a,s)), r'),$$
$$(a \in s \wedge \mathit{uses}(a) \equiv \mathit{true} \wedge \mathit{False} \wedge q(s,r)) \Rightarrow q(\mathit{insert}(\mathit{release}(a), \mathit{remove}(a,s)), r'),$$
...

variable elimination and Boolean rules

$$\vdash\ (a \in s \wedge q(s, \mathit{free})) \Rightarrow q(\mathit{insert}(a, \mathit{remove}(a,s)), \mathit{used}),$$
$$\mathit{True},$$
$$\mathit{True},$$
...

unfolding of q and Boolean rules

$$\vdash\ (a \in s \wedge |\mathit{filter}(\mathit{uses}, s)| \equiv 0) \Rightarrow |\mathit{filter}(\mathit{uses}, \mathit{insert}(a, \mathit{remove}(a,s)))| \le 1,$$
...

expansion with $|\mathit{filter}(\mathit{uses}, s)| \equiv 0) \Rightarrow |\mathit{filter}(\mathit{uses}, \mathit{insert}(a, \mathit{remove}(a,s)))| \le 1$

$$\vdash\ (a \in s \wedge |\mathit{filter}(\mathit{uses}, s)| \equiv 0) \Rightarrow |\mathit{filter}(\mathit{uses}, s)| \equiv 0,$$
...

Boolean rules

$$\vdash\ (r \stackrel{e/act}{\longrightarrow} r' \wedge s \stackrel{act}{\Longrightarrow} s' \wedge q(s,r)) \Rightarrow q(s', r'), \quad \textbf{(2)}$$
...

unfold $\longrightarrow$ (with axioms D,E)

$$\vdash\ (r \equiv \mathit{free} \wedge e \equiv i.\mathit{Request} \wedge \mathit{act} \equiv i.\mathit{Access} \wedge r' \equiv \mathit{used} \wedge s \stackrel{act}{\Longrightarrow} s' \wedge q(s,r)) \Rightarrow q(s', r'),$$
...

variable elimination

$$\vdash\ (s \stackrel{i.Access}{\Longrightarrow} s' \wedge q(s,r)) \Rightarrow q(s', \mathit{used}),$$
...

unfold $\Longrightarrow$ (with axioms F,G)

$$\vdash\ (s' \equiv \mathit{insert}(a', \mathit{remove}(a,s)) \wedge a \in s \wedge a \stackrel{Access}{\longrightarrow} a' \wedge id(a) \equiv i) \wedge q(s,r)) \Rightarrow q(s', \mathit{used}),$$
...

variable elimination

$$\vdash\ (a \in s \wedge a \stackrel{Access}{\longrightarrow} a' \wedge q(s,r)) \Rightarrow q(\mathit{insert}(a', \mathit{remove}(a,s)), \mathit{used}),$$
...

unfold q

$$\vdash\ (a \in s \wedge a \stackrel{Access}{\longrightarrow} a' \wedge |\mathit{filter}(\mathit{uses}, s)| \equiv 0) \Rightarrow |\mathit{filter}(\mathit{uses}, \mathit{insert}(a', \mathit{remove}(a,s)))| \le 1,$$
...

expansion with $|\mathit{filter}(\mathit{uses}, s)| \equiv 0) \Rightarrow |\mathit{filter}(\mathit{uses}, \mathit{insert}(a', \mathit{remove}(a,s)))| \le 1$

$$\vdash\ (a \in s \wedge a \stackrel{Access}{\longrightarrow} a' \wedge |\mathit{filter}(\mathit{uses}, s)| \equiv 0) \Rightarrow |\mathit{filter}(\mathit{uses}, s)| \equiv 0,$$
...

Boolean rules

$\vdash \quad (s \overset{i.e}{\Longrightarrow} s' \wedge r \overset{e}{\longrightarrow} r' \wedge q(s,r) \Rightarrow q(s',r'),$ **(3)**

$\dots$

unfold $\Longrightarrow$ (with axioms F,G)

$\vdash \quad (s' \equiv insert(a', remove(a,s)) \wedge a \in s \wedge a \overset{e}{\longrightarrow} a' \wedge id(a) \equiv i \wedge r \overset{e}{\longrightarrow} r' \wedge q(s,r)) \Rightarrow q(s',r'),$

$\dots$

variable elimination

$\vdash \quad (a \in s \wedge a \overset{e}{\longrightarrow} a' \wedge r \overset{e}{\longrightarrow} r' \wedge q(s,r) \Rightarrow q(insert(a', remove(a,s)), r'),$

$\dots$

unfold $\longrightarrow$ (with axioms A,B,C)

$$\begin{aligned}\vdash \quad &(a \in s \wedge \\ &((e \equiv Request \wedge a' \equiv a) \vee (e \equiv Access \wedge a' \equiv access(a)) \vee \\ &(e \equiv Release \wedge a' \equiv release(a) \wedge uses(a) \equiv true)) \\ &\wedge r \overset{e}{\longrightarrow} r' \wedge q(s,r)) \\ &\quad \Rightarrow q(insert(a', remove(a,s)), r'),\end{aligned}$$

$\dots$

Boolean rules and variable elimination

$\vdash \quad (a \in s \wedge r \overset{Request}{\longrightarrow} r' \wedge q(s,r)) \Rightarrow q(insert(a, remove(a,s)), r'),$ **(10)**

$(a \in s \wedge r \overset{Access}{\longrightarrow} r' \wedge q(s,r)) \Rightarrow q(insert(access(a), remove(a,s)), r')),$ **(11)**

$(a \in s \wedge uses(a) \equiv true \wedge r \overset{Release}{\longrightarrow} r' \wedge q(s,r)) \Rightarrow q(insert(release(a), remove(a,s)), r')),$ **(12)**

$\dots$

unfold $\longrightarrow$ (with axioms D,E)

$\vdash \quad (a \in s \wedge False \wedge q(s,r)) \Rightarrow q(insert(a, remove(a,s)), r'),$

$(a \in s \wedge False \wedge q(s,r)) \Rightarrow q(insert(access(a), remove(a,s)), r'),$

$(a \in s \wedge uses(a) \equiv true \wedge r \equiv used \wedge r' \equiv free \wedge q(s,r)) \Rightarrow q(insert(release(a), remove(a,s)), r'),$

$\dots$

Boolean rules and variable elimination

$\vdash \quad True,$

$True,$

$(a \in s \wedge uses(a) \equiv true \wedge q(s, used)) \Rightarrow q(insert(release(a), remove(a,s)), free),$

$\dots$

Boolean rules and unfolding of q

$$\begin{aligned}\vdash \quad &(a \in s \wedge uses(a) \equiv true \wedge |filter(uses,s)| \leq 1) \\ &\quad \Rightarrow |filter(uses, insert(release(a), remove(a,s)))| \equiv 0,\end{aligned}$$

$\dots$

expansion with $filter(uses, insert(release(a), s)) \sim filter(uses, s)$

$\vdash \quad (a \in s \wedge uses(a) \equiv true \wedge |filter(uses,s)| \leq 1) \Rightarrow |filter(uses, remove(a,s))| \equiv 0,$

$\dots$

expansion with $(a \in s \wedge uses(a) \equiv true \wedge |filter(uses,s)| \leq 1) \Rightarrow |filter(uses, remove(a,s))| \equiv 0$

$$\begin{aligned}\vdash \quad &(a \in s \wedge uses(a) \equiv true \wedge |filter(uses,s)| \leq 1) \\ &\quad \Rightarrow (a \in s \wedge uses(a) \equiv true \wedge |filter(uses,s)| \leq 1),\end{aligned}$$

$\dots$

Boolean rules

$\vdash \quad (s \overset{New(i)}{\Longrightarrow} s' \wedge q(s,r)) \Rightarrow q(s',r)$ **(4)**

unfold $\Longrightarrow$ (with axioms F,G)

$\vdash \quad s' \equiv insert(new(i), s) \wedge q(s,r)) \Rightarrow q(s',r)$

variable elimination

$\vdash \quad q(s,r) \Rightarrow q(insert(new(i), s), r)$

unfold q

$\vdash \; ((r \equiv free \wedge |filter(uses,s)| \equiv 0) \;\vee\; (r \equiv used \wedge |filter(uses,s)| \leq 1))$
$\quad \Rightarrow \; ((r \equiv free \wedge |filter(uses, insert(new(i),s))| \equiv 0) \;\vee$
$\quad\quad (r \equiv used \wedge |filter(uses, insert(new(i),s))| \leq 1))$

expansion with $filter(uses, insert(new(i),s)) \sim filter(uses,s)$

$\vdash \; ((r \equiv free \wedge |filter(uses,s)| \equiv 0) \;\vee\; (r \equiv used \wedge |filter(uses,s)| \leq 1))$
$\quad \Rightarrow \; ((r \equiv free \wedge |filter(uses,s)| \equiv 0) \;\vee\; (r \equiv used \wedge |filter(uses,s)| \leq 1))$

Boolean rules

$\vdash \; True$ ❑

Example 6.2.5 We extend RESERVATION (cf. Ex. 6.1.2) by the specification of a state machine that establishes a communication between *Ticket*- and *Customer*-objects.

RESMACHINE = RESERVATION `and` FINSET `then`

`sorts` $event \quad label$

`hidsorts` $CT = Customer + Ticket$

`constructs`
$New : CustomId \times String \rightarrow event$
$New : TicketId \rightarrow event$
$Order : CustomId \times Date \times (int+1) \times String \rightarrow event$
$Sell : TicketId \times CustomId \rightarrow event$
$\_.\_ : String \times event \rightarrow event$
$\_.\_ : int \times event \rightarrow event$
$\_ : event \rightarrow label$
$\_/\_ : event \times event \rightarrow label$
$sold : \rightarrow label$

`dynamic preds`
$\_ \xrightarrow{\_} \_ : Customer \times label \times Customer$
$\_ \xrightarrow{\_} \_ : Ticket \times label \times Ticket$
$\_ \overset{\_}{\Longrightarrow} \_ : set(CT) \times label \times set(CT)$

`vars`
$c, c' : Customer \quad cid : CustomId \quad t, t' : Ticket \quad tid : TicketId \quad pho, sea : String$
$dat : Date \quad i : int+1 \quad e, a : event$
$s, s_1, s_2 : set(CT) \quad dat : Date \quad i : int \quad tim : TimeOfDay$

`Horn axioms`
$c \xrightarrow{Order(cid,dat,i,sea)} c \;\Leftarrow\; name(c) \equiv cid$
$c \xrightarrow{Sell(tid,cid)} buy(c,t) \;\Leftarrow\; ID(t) \equiv tid \wedge name(c) \equiv cid$
$t \xrightarrow{Order(cid,dat,i,sea)/Sell(tid,cid)} sell(t,c)$
$\quad \Leftarrow \; ID(t) \equiv tid \wedge available(t) \equiv true \wedge name(c) \equiv cid$
$s \overset{New(cid,pho)}{\Longrightarrow} insert(\kappa_1(new(cid,pho)),s)$
$s \overset{New(tid)}{\Longrightarrow} insert(\kappa_2(new(tid)),s)$

(A) $s \overset{e}{\Longrightarrow} insert(\kappa_1(c'), remove(\kappa_1(c),s)) \;\Leftarrow\; c \xrightarrow{e} c' \;\wedge\; name(c) \equiv cid$

(B) $s \overset{e/a}{\Longrightarrow} insert(\kappa_2(t'), remove(\kappa_2(t),s)) \;\Leftarrow\; t \xrightarrow{e/a} t' \;\wedge\; ID(t) \equiv tid$

(C) $s \overset{sold}{\Longrightarrow} s_2 \;\Leftarrow\; s \overset{e/a}{\Longrightarrow} s_1 \;\wedge\; s_1 \overset{a}{\Longrightarrow} s_2$

Since the only action *cid.Buy*(*tid*) also occurs as an event, we did not introduce a particular sort for actions. Here the class-scope operation *new* operates on *sets* of objects and thus induces events that trigger transitions between states of several objects. Axioms A and B describe how single-object transitions lead to multiple-object transitions. Axiom C establishes a communication between objects. Similar transition relations are part of the command language specification given in Section 3.4. ❑

Example 6.2.6 (dining philosophers) In Example 5.2.1 we have presented a Petri net specification of this example. Here is a state machine specification designed along the lines of the preceding example, but with the main hidden sorts (*phil* and *event*) specified only in terms of object constructors.[12]

PHILS = INT `and` BOOL `and` FINSET `then`

`sorts` $event \quad label$

`hidsorts` $phil \quad fork \quad environment = set(phil) \times set(fork)$

`constructs`
$Ph, Fo : int \to event$
$Get, Put :\to event$
$GetLF, PutLF, GetRF, PutRF :\to event$
$\_.\_ : int \times event \to event$
$\_ : event \to label$
$\_/\_ : event \times event \to label$

`objconstructs`
$ph : int \to phil$
$fo : int \to fork$
$think, waitForEating, eat, waitForThinking : phil \to phil$
$taken, available : fork \to fork$

`defuncts`
$pid : phil \to int$
$fid : fork \to int$
$max :\to int$
$LF, RF : phil \to fork$

`dynamic preds`
$\_ \xrightarrow{l} \_ : phil \times label \times phil$
$\_ \xrightarrow{l} \_ : fork \times label \times fork$
$\_ \xRightarrow{l} \_ : set(phil) \times label \times set(phil)$
$\_ \xRightarrow{l} \_ : set(fork) \times label \times set(fork)$
$\_ \xRightarrow{l} \_ : environment \times label \times environment$

ν`-preds`
$P_1, P_2, P_3 : environment$
$\Box r : environment$ for all $r : environment$

`vars`
$i : int \quad p, p' : phil \quad f, f' : fork \quad ps, ps' : set(phil) \quad fs, fs' : set(fork) \quad e, a : event$
$l : label \quad env, env' : environment$

`Horn axioms`
$pid(ph(i)) \equiv i$
$fid(fo(i)) \equiv i$
$LF(ph(i)) \equiv fo(i)$
$RF(ph(i)) \equiv fo(i+1) \quad \Leftarrow \quad i < max$
$RF(ph(max)) \equiv fo(1)$
$pid(c(p)) \equiv pid(p)$ for all object constructors $c : phil \to phil$
$fid(c(f)) \equiv fid(p)$ for all object constructors $c : fork \to fork$
$think(p) \xrightarrow{GetLF} waitForEating(p)$
$waitForEating(p) \xrightarrow{GetRF} eat(p)$
$eat(p) \xrightarrow{PutRF} waitForThinking(p)$
$waitForThinking(p) \xrightarrow{PutLF} think(p)$
$available(f) \xrightarrow{Get} taken(f)$
$taken(f) \xrightarrow{Put} available(f)$
$ps \xRightarrow{Ph(i)} insert(ph(i), ps)$
$fs \xRightarrow{Fo(i)} insert(fo(i), fs)$
$ps \xRightarrow{i.a} insert(p', remove(p, ps)) \quad \Leftarrow \quad p \xrightarrow{a} p' \ \wedge \ id(p) \equiv i$
$fs \xRightarrow{i.e} insert(f', remove(f, fs)) \quad \Leftarrow \quad f \xrightarrow{e} f' \ \wedge \ id(f) \equiv i$

[12] The specification has been inspired by the introductory example of [59].

$$
\begin{array}{lll}
& (ps, fs) \stackrel{a}{\Longrightarrow} (ps', fs') \quad \Leftarrow \quad ps \stackrel{a}{\Longrightarrow} s' \ \wedge \ fs \stackrel{a}{\longrightarrow} fs' & \\
\texttt{co-Horn axioms} & P_1(ps, fs) \ \Rightarrow \ (p \in ps \ \Rightarrow \ \exists p' : (p \equiv think(p') \ \dot{\vee} \ p \equiv eat(p'))) & \\
& P_2(ps, fs) \ \Rightarrow \ (f \in fs \ \Rightarrow \ \exists f' : (f \equiv available(f') \ \dot{\vee} \ f \equiv taken(f'))) & \\
& P_3(ps, fs) \ \Rightarrow \ ((f \in fs \wedge p \in ps \wedge LF(p) \equiv f) & \\
& \qquad \Rightarrow \ \exists f' : (f \equiv available(f') \ \dot{\vee} \ p \not\equiv think(p'))) & \\
& P_3(ps, fs) \ \Rightarrow \ ((f \in fs \wedge p \in ps \wedge RF(p) \equiv f) & \\
& \qquad \Rightarrow \ \exists f' : (f \equiv available(f') \ \dot{\vee} \ p \not\equiv think(p'))) & \\
& \Box r(env) \ \Rightarrow \ r(env) & \\
& \Box r(env) \ \Rightarrow \ (env \stackrel{l}{\Longrightarrow} env' \Rightarrow \Box r(env')) &
\end{array}
$$

Exercise. In terms of PHILS, the three correctness conditions of Example 5.3.2 are $\Box P_1(\emptyset, \emptyset)$, $\Box P_2(\emptyset, \emptyset)$ and $\Box P_3(\emptyset, \emptyset)$. Show that they are inductive theorems of PHILS. ❏

References

[1] M.A. Arbib, E.G. Manes, *Arrows, Structures, Functors: The Categorical Imperative*, Academic Press 1975

[2] M.A. Arbib, E.G. Manes, *Parametrized Data Types Do Not Need Highly Constrained Parameters*, Information and Control 52 (1982) 139-158

[3] E. Astesiano and G. Reggio, *Labelled Transition Logic: An Outline*, Technical Report DISI-TR-96-20, University of Genova 1996

[4] E. Astesiano, H.-J. Kreowski, B. Krieg-Brückner, eds., *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Report, Springer 1999

[5] E. Astesiano, M. Broy, G. Reggio, *Algebraic Specification of Concurrent Systems*, in [4]

[6] E. Astesiano, G. Reggio, *Algebraic Specification of Concurrency*, Proc. WADT'91, Springer LNCS 655 (1993) 1-39

[7] J.C.M. Baeten, W.P. Weijland, *Process Algebra*, Cambridge University Press 1990

[8] M. Barr, *Terminal Coalgebras in Well-founded Set Theory*, Theoretical Computer Science 114 (1993) 299-315

[9] K.A. Bartlet, R.A. Scantlebury, P.T. Wilkinson, *A Note on Reliable Full-Duplex Transmission over Half-Duplex Links*, Comm. ACM 12 (1969) 260-261

[10] J. van Benthem, *Exploring Logical Dynamics*, Cambridge University Press 1996

[11] R. Bird, *Introduction to Functional Programming using Haskell*, Prentice Hall 1998

[12] M. Broy, M. Wirsing, *Partial Abstract Types*, Acta Informatica 18 (1982) 47-64

[13] M. Broy, *Requirement and Design Specification for Distributed Systems: The Lift Problem*, Proc. Workshop on Future Trends of Distributed Computing Systems in the 1990s, IEEE Computer Society Press (1988) 164-173

[14] R.E. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Transactions on Computers 35 (1986) 677-691

[15] L. Cardelli, A.D. Gordon, *Anytime, Anywhere: Modal Logics for Mobile Ambients*, Proc. POPL 2000, ACM Press (2000) 365-377

[16] CCITT, *Functional Specification and Description Language SDL*, CCITT Red Book, Vol. 6, Recommendations Z.100-Z.104, CCITT, Geneva 1984

[17] E.F. Codd, *A Relational Model for Large Shared Data Banks*, Communications of the ACM 13 (1970) 377-387

[18] The CoFI Task Group on Language Design, *CASL: The Common Algebraic Specification Language*, 1998, www.brics.dk/Projects/CoFI/Documents/CASL/Summary

[19] B. Demuth, H. Hußmann, *Using UML/OCL Constraints for Relational Database Design*, Proc. UML '99, 1999

[20] L. Dennis, A. Bundy, I. Green, *Using a Generalisation Critic to Find Bisimulations for Coinductive Proofs*, Proc. CADE 14, Springer LNAI 1249 (1997) 276-290

[21] L. Dennis, *Proof Planning Coinduction*, Ph.D. thesis, University of Edinburgh 1998

[22] J. Desel, E. Kindler, *Proving Correctness of Distributed Algorithms: A Petri Net Approach*, AIFB-Bericht 348, University of Karlsruhe, 1997,
www.informatik.hu-berlin.de/∼kindler/PostScript/AIFB348.ps

[23] H.-D. Ehrich, *Object Specification*, in [4]

[24] M. Erwig, *Categorical Programming with Abstract Data Types*, Proc. AMAST '98, Springer LNCS 1548 (1998) 406-421

[25] M.M. Fokkinga, *Datatype Laws without Signatures*, Math. Structures in Comp. Sci. 6 (1996) 1-32

[26] R. Giegerich, *A Systematic Approach to Dynamic Programming in Bioinformatics. Parts 1 and 2: Sequence Comparison and RNA Folding*, Report 99-05, Technische Fakultät, Universität Bielefeld 1999

[27] R. Giegerich, C. Meyer, *Algebraic Dynamic Programming*, Proc. AMAST 2002, Springer LNCS 2422 (2002) 249-364

[28] R. Giegerich, P. Steffen, *Implementing Algebraic Dynamic Programming in the Functional and the Imperative Programming Paradigm*, Proc. MPC 2002, Springer LNCS 2386 (2002) 1-20

[29] M. Gogolla, M. Richters, *Transformation Rules for UML Class Diagrams*, Proc. UML '98, Springer LNCS 1618 (1998) 92-106

[30] J.A. Goguen, R. Diaconescu, *An Oxford Survey of Order Sorted Algebra*, Mathematical Structures in Computer Science 4 (1994) 363-392

[31] J.A. Goguen, K. Lin, G. Roşu, *Conditional Circular Coinduction*, UCSD Report, San Diego 2003, www-cse.ucsd.edu/users/goguen/ps/c4rw.ps

[32] J.A. Goguen, G. Malcolm, *A Hidden Agenda*, UCSD Technical Report CS97-538, San Diego 1997, www-cse.ucsd.edu/users/goguen/ps/ha.ps.gz

[33] J.A. Goguen, J. Meseguer, *Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics*, in: B. Shriver, P. Wegner, eds., Research Directions in Object-Oriented Programming, MIT Press (1987) 417-477

[34] J.A. Goguen, *Stretching First Order Equational Logic: Proofs with Partiality, Subtypes and Retracts*, UCSD Report, San Diego 1997, www-cse.ucsd.edu/users/goguen/ps/ftp97.ps.gz

[35] J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright, *Initial Algebra Semantics and Continuous Algebras*, J. ACM 24 (1977) 68-95

[36] T. Hagino, *Codatatypes in ML*, J. Symbolic Computation 8 (1989) 629-650

[37] A. Hamie, F. Civello, J. Howe, S. Kent, R. Mitchell, *Reflections on the Object Constraint Language*, Proc. UML '98, 1998

[38] M. Hanus, ed., *Curry: An Integrated Functional Logic Language*, RWTH Aachen, 1999, www-i2.informatik.rwth-aachen.de/∼hanus/curry/papers/report.dvi.Z

[39] M. Hanus, *Distributed Programming in Curry*, Proc. WFLP '99, IMAG Report RR 1021-I-, University of Grenoble (1999) 195-208

[40] M. Hanus, *Server Side Web Scripting in Curry*, Proc. LPSE 2000, IMAG Report RR 1021-I-, University of Grenoble (1999) 195-208

[41] D. Harel, M. Politi, *Modeling Reactive Systems With Statecharts: The STATEMATE Approach*, McGraw-Hill 1998

[42] M. Hennessy, R. Milner, *Algebraic Laws for Nondeterminism and Concurrency*, J. ACM 32 (1985) 137-161

[43] R. Hennicker, A. Kurz, *(Ω, Ξ)-Logic: On the Algebraic Extension of Coalgebraic Specifications*, Proc. CMCS '99, Elsevier ENTCS 19 (1999) 195-211

[44] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall 1985

[45] P. Hudak, J. Peterson, J.H. Fasel, *A Gentle Introduction to Haskell 98*, Report, 1999, see www.haskell.org

[46] H. Hußmann, M. Cerioli, G. Reggio, F. Tort, *Abstract Data Types and UML Models*, Report DISI-TR-99-15, University of Genova 1999

[47] H. Hußmann, B. Demuth, F. Finger, *Modular Architecture for a Toolset Supporting OCL*, Proc. UML 2000, Springer LNCS 1939 (2000) 278-293

[48] S. Janson, J. Montelius, S. Haridi, *Ports for Objects in Concurrent Logic Programs*, in: G. Agha, P. Wegner, A. Yonezawa, eds., Research Directions in Object-Based Concurrency, MIT Press (1993) 211-231

[49] B. Jacobs, J. Rutten, *A Tutorial on (Co)Algebras and (Co)Induction*, EATCS Bulletin 62 (1997) 222-259

[50] K. Jensen, *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, Vol. 1-3, Springer 1995-1997

[51] K. Jensen, *A Brief Introduction to Coloured Petri Nets*, Aarhus 1997

[52] E. Kindler, W. Reisig, *Algebraic System Nets for Modelling Distributed Algorithms*, Petri Net Newsletter 51 (1996) 16-31

[53] E. Kindler, W. Reisig, *Verification of Distributed Algorithms with Algebraic Petri Nets*, in: C. Freksa, M. Jantzen, R. Valk, eds., Foundations of Computer Science: Potential - Theory - Cognition, Springer LNCS 1337 (1997) 261-270

[54] E. Kindler, H. Völzer, *Flexibility in Algebraic Nets*, Informatik-Bericht Nr. 89, Humboldt-Universität Berlin 1997

[55] U. Kühler, C.-P. Wirth, *Conditional Equational Specifications of Data Types with Partial Operations for Inductive Theorem Proving*, Proc. RTA '97, Springer LNCS 1232 (1997) 38-52

[56] U. Lechner, C. Lengauer, M. Wirsing *An Object-Oriented Airport: Specification and Refinement in Maude*, Proc. WADT '94, Springer LNCS 906 (1995) 351-367

[57] D.J. Lehmann, M.B. Smyth, *Algebraic Specification of Data Types: A Synthetic Approach*, Mathematical Systems Theory 14 (1981) 97-139

[58] J. Lilius, I.P. Paltor, *The Semantics of UML State Machines*, TUCS Technical Report 273, Turku Centre for Computer Science 1999

[59] J. Lilius, I.P. Paltor, *vUML: A Tool for Verifying UML Models*, TUCS Technical Report 272, Turku Centre for Computer Science 1999

[60] G. Malcolm, *Behavioural Equivalence, Bisimulation, and Minimal Realisation*, Proc. WADT '95, Springer LNCS 1130 (1996) 359-378

[61] E.G. Manes, *Algebraic Theories*, Springer 1976

[62] M. Mandel, M.V. Cengarle, *On the Expressive Power of the Object Constraint Language OCL*, Proc. FM '99, Springer LNCS 1708 (1999) 854-874

[63] E.G. Manes, M.A. Arbib, *Algebraic Approaches to Program Semantics*, Springer 1986

[64] J. Meseguer, *A Logical Theory of Concurrent Objects and its Realization in the Maude Language*, in: G. Agha, P. Wegner, A. Yonezawa, eds., Research Directions in Object-Based Concurrency, MIT Press (1993) 313-389

[65] J. Meseguer, *Membership Algebra as a Logical Framework for Equational Specification*, Proc. WADT '97, Springer LNCS 1376 (1998) 18-61

[66] J. Meseguer, J.A. Goguen, *Initiality, Induction and Computability*, in: M. Nivat, J. Reynolds, eds., Algebraic Methods in Semantics, Cambridge University Press (1985) 459-541

[67] R. Milner, *Communication and Concurrency*, Prentice-Hall 1989

[68] R. Milner, *A Communicating and Mobile Systems: the π-Calculus*, Cambridge University Press 1999

[69] E. Moggi, *Notions of Computation and Monads*, Information and Computation 93 (1991) 55-92

[70] M. Müller-Olm, D. Schmidt, B. Steffen, *Model Checking: A Tutorial Introduction*, Proc. SAS '99, Springer LNCS 1694 (1999) 330-394

[71] L.C. Paulson, *ML for the Working Programmer*, 2nd edition, Cambridge University Press 1996

[72] P. Padawitz, *Deduction and Declarative Programming*, Cambridge University Press 1992

[73] P. Padawitz, *Inductive Theorem Proving for Design Specifications*, J. Symbolic Computation 21 (1996) 41-99

[74] P. Padawitz, *Proof in Flat Specifications*, in [4]

[75] P. Padawitz, *Swinging Types = Functions + Relations + Transition Systems*, Theoretical Computer Science 243 (2000) 93-165

[76] P. Padawitz, *Theorie der Programmierung*, Course Notes, University of Dortmund, http://ls5.cs.uni-dortmund.de/∼peter/TdP96.ps.gz

[77] P. Padawitz, *Swinging UML: How to Make Class Diagrams and State Machines Amenable to Constraint Solving and Proving*, Proc. UML 2000, Springer LNCS 1939 (2000) 162-177

[78] P. Padawitz, *Structured Swinging Types*, Report, University of Dortmund 2003, ls5-www.cs.uni-dortmund.de/∼peter/SST.ps.gz

[79] P. Padawitz, *Dialgebraic Swinging Types*, Report, University of Dortmund 2003, ls5-www.cs.uni-dortmund.de/∼peter/Dialg.ps.gz

[80] P. Padawitz, *Expander2: A Formal Methods Presenter and Animator*, ls5-www.cs.uni-dortmund.de/∼peter/Expander2/Expander2.html

[81] P. Padawitz, *Expander2: Towards a Workbench for Interactive Formal Reasoning*, ls5-www.cs.uni-dortmund.de/∼peter/Chiemsee.ps.gz

[82] J. Peterson et al., *Haskell: A Non-strict, Purely Functional Language (Version 1.4)*, Report, Yale University 1997

[83] S.L. Peyton Jones, P. Wadler, *Imperative Functional Programming*, Proc. POPL '93, ACM Press (1993) 71-84,
http://research.microsoft.com/Users/simonpj/Papers/imperative.ps.Z

[84] G.D. Plotkin, *An Operational Semantics for CSP*, in: D. Bjørner, ed., Proc. IFIP TC-2 Working Conf. Formal Description of Programming Concepts II, North-Holland (1983) 199-225

[85] H. Reichel, *An Approach to Object Semantics based on Terminal Coalgebras*, Math. Structures in Comp. Sci. 5 (1995) 129-152

[86] H. Reichel, *Unifying ADT- and Evolving Algebra Specifications*, EATCS Bulletin 59 (1996) 112-126

[87] W. Reisig, *Petri Nets and Algebraic Specifications*, Theoretical Computer Science 80 (1991) 1-34

[88] W. Reisig, *Elements of Distributed Algorithms: Modeling and Verification with Petri Nets*, Springer 1997

[89] M. Richters, M. Gogolla, *On Formalizing the UML Object Constraint Language OCL*, in: Proc. Conceptual Modeling- ER '98, Springer LNCS 1507 (1998) 449-464

[90] M. Richters, M. Gogolla, *Validating UML Models and OCL Constraints*, in: Proc. UML 2000, Springer LNCS 1939 (2000) 265-277

[91] G. Roşu, J.A. Goguen, *Hidden Congruent Deduction*, UCSD Report, San Diego 1999, www-cse.ucsd.edu/users/goguen/ps/cong.ps.gz

[92] G. Roşu, J.A. Goguen, *Circular Coinduction*, UCSD Report, San Diego 1999, www-cse.ucsd.edu/users/goguen/ps/circ.ps.gz

[93] J. Rumbaugh, I. Jacobson, G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley 1999

[94] J.J.M.M. Rutten, *Universal Coalgebra: A Theory of Systems*, Report CS-R9652, CWI, SMC Amsterdam 1996

[95] J.J.M.M. Rutten, D. Turi, *Initial Algebra and Final Coalgebra Semantics for Concurrency*, Report CS-R9409, CWI, SMC Amsterdam 1994

[96] J. Setubal, J. Meidanis, *Computational Molecular Biology*, PWS Publishing Company 1997

[97] E. Shapiro, A. Takeuchi, *Object-Oriented Programming in Concurrent Prolog*, in: E. Shapiro, ed., Concurrent Prolog: Collected Papers Vol. 2, MIT Press (1987) 251-273

[98] C. Stirling, *Modal and Temporal Logics*, in: S. Abramsky et al., eds., Handbook of Logic in Computer Science, Clarendon Press (1992) 477-563

[99] C. Stirling, D. Walker, *Local Model Checking in the Modal μ-Calculus*, Proc. TAPSOFT '89, Springer LNCS 351 (1989) 369-383

[100] K.J. Turner, ed., *Using Formal Description Techniques: An Introduction to Estelle, Lotos and SDL*, John Wiley & Sons 1993

[101] P. Wadler, *Monads for Functional Programming*, in: M. Broy, ed., Program Design Calculi, Springer 1993

[102] J.B. Warmer, A.G. Kleppe, *The Object Constraint Language*, Addison-Wesley 1999

[103] M. Weber, R. Walter, H. Völzer, T. Vesper, W. Reisig, S. Peuker, E. Kindler, J. Freiheit, J. Desel, *DAWN: Petrinetzmodelle zur Verifikation Verteilter Algorithmen*, Informatik-Bericht Nr. 88, Humboldt-Universität Berlin 1997

[104] P. Wegner, *Why Interaction Is More Powerful Than Algorithms*, Communications of the ACM 40 (1997) May, 80-91